Automatische detectie van recursiepatronen

Jasper Van der Jeugt

Promotor: prof. Tom Schrijvers
Begeleider: Steven Keuchel

Masterproef ingediend tot het behalen van de academische graad van
Master in de ingenieurswetenschappen: computerwetenschappen

Vakgroep Toegepaste Wiskunde, Informatica en Statistiek
Voorzitter: prof. dr. Willy Govaerts
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2012-2013

Automatische detectie van recursiepatronen

Jasper Van der Jeugt

Promotor: prof. Tom Schrijvers
Begeleider: Steven Keuchel

Masterproef ingediend tot het behalen van de academische graad van
Master in de ingenieurswetenschappen: computerwetenschappen

Vakgroep Toegepaste Wiskunde, Informatica en Statistiek
Voorzitter: prof. dr. Willy Govaerts
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2012-2013

Toelating tot bruikleen

De auteur(s) geeft(geven) de toelating deze masterproef voor consultatie beschikbaar te stellen en delen van de masterproef te kopiëren voor persoonlijk gebruik. Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de 2verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze masterproef.

The author(s) gives (give) permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In the case of any other use, the limitations of the copyright have to be respected, in particular with regard to the obligation to state expressly the source when quoting results from this master dissertation.

Jasper Van der Jeugt
mei 2013

Dankwoord

Het schrijven van een thesis is een uitputtende en tijdrovende zaak. Een groot aantal mensen hebben mij echter bijgestaan en ervoor gezorgd dat ik zelfs in moeilijke perioden steeds mijn motivatie en interesse bleef behouden.

Dank gaat uit naar mijn promotor, prof. dr. ir. Tom Schrijvers en mijn begeleider Steven Keuchel. Voor vele correcties in de thesistekst wil ik graag Andy Georges, Tom Naessens en Toon Willems bedanken. Ten laatste wil ik mijn ouders bedanken, en natuurlijk ook mijn vrienden, voor de schitterende studententijd die ik op de Universiteit Gent beleefd heb.

Jasper Van der Jeugt
mei 2013

Automatische detectie van recursiepatronen

Jasper Van der Jeugt
Promotor: prof dr. ir. Tom Schrijvers
Begeleider: Steven Keuchel

In functionele programmeertalen brengt het herschrijven van expliciet recursieve functies naar functies in termen van hogere-orde functies zoals *fold* en *map* veel voordelen zoals beknoptheid en leesbaarheid. Bovendien maakt het bepaalde optimalisaties mogelijk. Helaas is het voor een programmeur niet altijd eenvoudig om functies in deze stijl te schrijven. Wij stellen een aanpak voor die deze hogere-orde functies automatisch kan detecteren en de code transformeren, zodat de programmeur de voordelen van deze optimalisaties krijgt, ongeacht in welke stijl hij of zij code schrijft.

Trefwoorden: catamorfismes, foldr/build-fusion, code-analysis, transformaties, functionele programmeertalen

Automatic Detection of Recursion Patterns

Jasper Van der Jeugt

Supervisor(s): prof dr. ir. Tom Schrijvers, Steven Keuchel

Abstract— **Rewriting explicitly recursive functions in terms of higher-order functions such as $fold$ and map brings many advantages such as conciseness, improved readability, and it facilitates some optimisations. However, it is not always straightforward for a programmer to write functions in this style. We present an approach to automatically detect these higher-order functions, so the programmer can have his cake and eat it, too.**

Keywords— **catamorphisms, fold-build fusion, analysis, transformation**

I. Introduction

Higher-order functions are immensely popular in Haskell, whose Prelude alone offers a wide range of them (e.g., *map*, $filter$, *any*, ...). This is not surprising, because they are the key *abstraction* mechanism of functional programming languages. They enable capturing and reusing common patterns among, often recursive, function definitions to a much larger extent than first-order functions. In addition to the obvious code reuse and increased programmer productivity, uses of higher-order functions have many other potential advantages over conventional first-order definitions.

- Uses of higher-order functions can be more quickly understood because they reduce the that is already known pattern to a single name and thus draw the attention immediately to what is new (i.e., the function parameters).
- Because the code is more compact and the number of bugs is proportional to code size [1], higher-order functions should lead to fewer bugs.
- Properties can be established for the higher-order function independently from its particular uses. This makes (e.g., equational) reasoning more productive.
- Since properties and occurrences are more readily available, they make good targets for automatic optimisation in compilers.

A particularly ubiquitous pattern is that of folds or *catamorphisms*. In the case of lists, this pattern has been captured in the well-known $foldr$ function.

$$
\begin{aligned}
&foldr :: (a \to b \to b) \to b \to [a] \to b \\
&foldr \_\ z\ [\,] \quad\ = z \\
&foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)
\end{aligned}
$$

Indeed, the research literature is full of applications, properties and optimisations based on folds [2, 3, 4, 5].

Hence, given all these advantages of folds, one would expect every programmer to diligently avoid explicit recursion where folds can do the job. Unfortunately, that is far from true in practice. For many reasons, programmers do not write their code in terms of explicit folds. This class comprises a large set of advanced functional programmers (see V-A). This is compounded by the fact that programmers often do not bother to define the equivalent of $foldr$ for other inductive algebraic datatypes.

Yet, sadly, these first-order recursive functions are treated as second-class by compilers. They do not benefit from the same performance gains like loop fusion and deforestation. In fact, the leading Haskell compiler GHC won't even inline recursive functions.

We disagree with this injustice and argue that it is quite unnecessary to penalize programmers for using first-class recursion. In fact, we show that with a little effort, catamorphisms can be detected automatically by the compiler and automatically transformed into explicit invocations of folds for the purpose of automation.

II. Folds, Builds and Fusion over Lists

Catamorphisms are functions that *consume* an inductively defined datastructure by means of structural recursion. They can be expressed in terms of $foldr$ [4].

The *build* function is the dual to the $foldr$ function. Where the $foldr$ function captures a pattern of list *consummation*, *build* captures a particular pattern of list *production*.

$$
\begin{aligned}
&build :: (\forall b.(a \to b \to b) \to b \to b) \to [a] \\
&build\ g = g\ (:)\ [\,]
\end{aligned}
$$

Given these functions, we can *fuse* the production and consummation of a list: meaning that no intermediate datastructure needs to be allocated. The foldr/build-fusion rule is given by:

$$foldr\ cons\ nil\ (build\ g) \quad \equiv \quad g\ cons\ nil$$

The main Haskell compiler, GHC, currently provides limited automation for foldr/build-fusion: by means of a GHC rewrite rule. When functions are defined explicitly in terms of $foldr$ and *build*, after inlining, the rewrite rule may perform the fusion.

Unfortunately, this existing automation is severely limited. Firstly, it is restricted to lists and secondly it requires programmers to explicitly define their functions in terms of *build* and $foldr$.

This work lifts both limitations. It allows programmers to write their functions in explicitly recursive style and performs foldr/build-fusion for any directly inductive datastructure.

III. Discovering Folds and Builds

We have devised a set of formal rewrite rules to turn explicitly recursive functions into folds and to rewrite producers into builds where possible.

A. Syntax and Notation

To simplify the presentation, we do not use Haskell source syntax or even GHC's core syntax (based on System F). Instead, we use the untyped lambda-calculus extended with constructors and pattern matching, and (possibly recursive) bindings.

$$
\begin{array}{lrcl}
\text{binding} & b & ::= & x = e \\
\text{pattern} & p & ::= & K\ \overline{x} \\
\text{expression} & e & ::= & x \\
 & & | & e\ e \\
 & & | & \lambda x \to e \\
 & & | & K \\
 & & | & \textbf{case e of } \overline{\textbf{p} \to \textbf{e}}
\end{array}
$$

The extension to GHC's full core syntax, including types, is relatively straightforward. We also need an advanced form of *(expression) context*:

$$
\begin{array}{rcl}
E & ::= & x \\
 & | & E\ x \\
 & | & E\ \square \\
 & | & E\ \triangle
\end{array}
$$

A context E denotes a function applied to a number of arguments. The function itself and some of its arguments are given (as variables), while there are holes for the other arguments. In fact, there are two kinds of holes: boxes $\square$ and triangles $\triangle$. The former is used for a sequence of unimportant arguments, while the latter marks an argument of significance.

The function $E[\overline{e}; e]$ turns a context E into an expression by filling in the holes with the given expressions.

$$
\begin{array}{lcl}
x[\epsilon; e] & = & x \\
(E\ x)[\overline{e}; e] & = & E[\overline{e}; e]\ x \\
(E\ \square)[\overline{e}, e_1; e] & = & E[\overline{e}; e]\ e_1 \\
(E\ \triangle)[\overline{e}; e] & = & E[\overline{e}; e]\ e
\end{array}
$$

Note that this function is partial; it is undefined if the number of expressions $\overline{e}$ does not match the number of box holes.

B. Rules

The non-deterministic rules to rewrite functions into folds or builds can be found in Figure 1 and Figure 2 respectively. To keep the exposition simple, they are only given for lists and not arbitrary algebraic datastructures: the specifics for this generalization can be found in the full text. As an illustration, consider *map*, which is both a fold as well as a build.

$$
\begin{array}{l}
map = \lambda f\ y \to \textbf{case y of} \\
\quad [\,] \quad\ \to [\,] \\
\quad (v : vs) \to f\ v : map\ f\ vs
\end{array}
$$

Applying the rules for build detection yields us:

$$
\begin{array}{l}
map = \lambda f \to \lambda l \to build\ (g\ f\ l) \\
g \quad = \lambda f \to \lambda l \to \lambda c \to \lambda n \to \\
\qquad \textbf{case l of} \\
\quad [\,] \quad\ \to n \\
\quad (y : ys) \to c\ (f\ y)\ (g\ f\ ys\ c\ n)
\end{array}
$$

Now g can be recognise by our fold detection rules:

$$
\begin{array}{l}
g = \lambda f \to \lambda l \to \lambda c \to \lambda n \to \\
\quad foldr\ (\lambda y\ ys \to c\ (f\ y)\ ys)\ n\ l
\end{array}
$$

IV. Implementation

We implemented these transformations on top of GHC [6], the de-facto standard compiler for Haskell. Since GHC 7.2.1, a plugin framework is available [7].

This plugin framework facilitates writing GHC Core [8] transformations. Using this framework, we no longer have to edit the GHC source code in order to add, modify or delete Core-to-Core passes: we can do so in a plugin. This plugin is then passed to GHC using a command-line argument.

We implemented such a plugin, which contains passes:

- A pass to convert functions written in explicitly recursive style into functions in terms of folds. This pass is a deterministic implementation of the rules in Figure 1. Directly inductive datastructures other than list are also supported.
- We have a similar pass for builds: a deterministic implementation of the rules in Figure 2.
- A custom inliner, over which we have a bit more control than the default GHC inliner. However, we ultimately decided not to use this custom inliner, so it is disabled by default.
- An implementation of foldr/build-fusion for any algebraic datatype for which we have a fold and a build. By using this pass, we can avoid having to add extra rule pragmas [9].

Additionally our work also contains Template Haskell [10] routines to mitigate the burden of defining folds and builds for algebraic datatypes. These folds and builds can be generated by issuing, e.g., for a type $Tree$:

```
$ (deriveFold ‘Tree "foldTree")
$ (deriveBuild ‘Tree "buildTree")
```

V. Evaluation

A. Identifying folds

A first aspect we can evaluate is how well our detection of folds works. Unfortunately, manually identifying folds in projects very labour-intensive. This explains why it is especially hard to detect false negatives.

Additionally, very little other related work is done. There is the *HLint* [11] tool is able to recognize folds as well, but its focus lies on refactoring rather than optimisations. We cannot directly compare our results with those of HLint since we detect folds over all possible directly inductive datastructures, and HLint is limited to list. Hence, we classify the folds we detect as either a fold over a list or a fold over some other datastructure.

Table I shows that we clearly detect more folds than the HLint tool does. Additionally, we also tried our tool

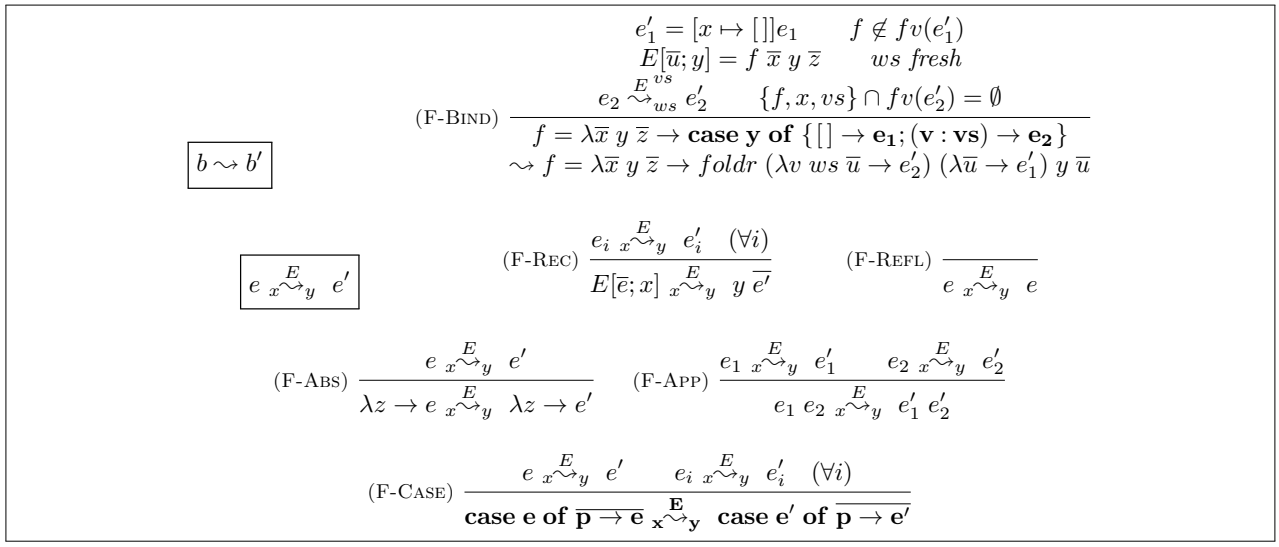


Fig. 1. Fold discovery rules

	Total	List	ADT	HLint
Cabal	20	11	9	9
containers	100	11	89	1
cpphs	5	2	3	1
darcs	66	65	8	6
ghc	327	216	111	26
hakyll	5	1	4	0
haskell-src-exts	37	11	26	2
hlint	6	3	3	0
hscolour	4	4	0	2
HTTP	6	6	0	3
pandoc	15	15	0	2
parsec	3	3	0	0
snap-core	4	3	1	0

TABLE I
Results of identifying folds in some well-known projects

	Total	List	ADT
Cabal	101	81	20
containers	25	2	23
cpphs	6	5	1
darcs	354	354	0
ghc	480	178	302
hakyll	22	18	4
haskell-src-exts	140	74	66
hlint	69	62	7
hscolour	33	33	0
HTTP	11	11	0
pandoc	97	97	0
parsec	10	10	0
snap-core	4	4	0

TABLE II
Results of identifying builds in some well-known projects

on the test cases included with HLint – which we could all correctly detect. This suggests that we detect a strict superset of possible folds, even for lists. The fact that the number of possible folds in these projects found by HLint is so low indicates that the authors of the respective packages might have used HLint during development.

B. Identifying builds

We also evaluated the detection of builds. However, as far as we know no work has been done in this area – hence, we cannot compare these results to anything meaningful. The results are shown in Table II. We conclude that they are within the same order of magnitude as the number of folds we found.

C. Foldr/build-fusion

Unfortunately it remains a hard problem to count how many times foldr/build-fusion can be applied in a Haskell package. The concrete difficulties we currently face are:

- Manual intervention is not required to do the detection of folds or builds, but in order to actually perform the transformation some manual changes are required: more precisely: adding imports and language pragmas, and using Template Haskell to derive the necessary folds.
- Code which is crucial to the performance of a package is often manually optimised. Hence, we usually find no opportunities to remove intermediate datastructures there.
- A lot of code is partially written using higher-order functions (e.g. from the *Data.List* library) and partially using

explicit recursion. Due to practical problems, we currently cannot perform any fusion there, although we are working to lift this limitation.

D. Benchmarks

We researched the speedups caused by foldr/build-fusion. In order to do this, we constructed small pipeline-like programs who have the inherent property of being fusable. Consider the functions:

$$
\begin{aligned}
&sumt :: Tree\ Int \to Int \\
&sumt\ (Leaf\ x) \quad\ = x \\
&sumt\ (Branch\ l\ r) = sumt\ l + sumt\ r \\
&mapt :: (a \to b) \to Tree\ a \to Tree\ b \\
&mapt\ f\ (Leaf\ x) \quad\ = Leaf\ (f\ x) \\
&mapt\ f\ (Branch\ l\ r) = Branch\ (mapt\ f\ l)\ (mapt\ f\ r) \\
&uptot :: Int \to Int \to Tree\ Int \\
&uptot\ lo\ hi \\
&\quad \mid lo \geqslant hi \quad\ = Leaf\ lo \\
&\quad \mid otherwise = \\
&\qquad Branch\ (uptot\ lo\ mid)\ (uptot\ (mid + 1)\ \mathrm{hi}) \\
&\ \ \textbf{where} \\
&\quad mid = (lo + hi)\ `div`\ 2
\end{aligned}
$$

These auxiliary functions allow us to write functions which must allocate intermediate datastructures, as the result of *uptot* and *mapt*. Hence, we can benchmark the functions:

$$
\begin{aligned}
&t1, t2, t3, t4, t5 :: Int \to Int \\
&t1\ n = sumt\ (1\ `\mathrm{uptot}`\ \mathrm{n}) \\
&t2\ n = sumt\ (mapt\ (+1)\ (1\ `\mathrm{uptot}`\ \mathrm{n})) \\
&t3\ n = sumt\ (mapt\ (+1)\ (\mathrm{mapt}\ (+1)\ (1\ `\mathrm{uptot}`\ \mathrm{n}))) \\
&t4\ n = \ldots \\
&t5\ n = \ldots
\end{aligned}
$$

$$
\boxed{b \rightarrowtail b'; b''}
$$

$$
\text{(B-Bind)}\ \frac{c, n, g \text{ fresh} \qquad e\ \overset{c,n}{f \rightarrowtail_g}\ e'}{f = \lambda \overline{x} \to e \quad \rightarrowtail \quad f = \lambda \overline{x} \to build\ (g\ \overline{x}); \quad g = \lambda \overline{x} \to \lambda c \to \lambda n \to e'}
$$

$$
\boxed{e\ \overset{c,n}{f \rightarrowtail_g}\ e'}
$$

$$
\text{(B-Rec)}\ \frac{}{f\ \overline{e}\ \overset{c,n}{f \rightarrowtail_g}\ g\ \overline{e}\ c\ n}
$$

$$
\text{(B-Nil)}\ \frac{}{[\,]\ \overset{c,n}{f \rightarrowtail_g}\ n} \qquad \text{(B-Cons)}\ \frac{e_2\ \overset{c,n}{f \rightarrowtail_g}\ e_2'}{(e_1 : e_2)\ \overset{c,n}{f \rightarrowtail_g}\ c\ e_1\ e_2'}
$$

$$
\text{(B-Build)}\ \frac{}{build\ e\ \overset{c,n}{f \rightarrowtail_g}\ e\ c\ n}
$$

$$
\text{(B-Case)}\ \frac{e_i\ \overset{c,n}{f \rightarrowtail_g}\ e_i' \quad (\forall i)}{\textbf{case e of } \overline{\textbf{p} \to \textbf{e}}\ \overset{\mathbf{c,n}}{\mathbf{f \rightarrowtail_g}}\ \textbf{case e of } \overline{\textbf{p} \to \textbf{e}'}}
$$

Fig. 2. Build discovery rules

The results can be found in Figure 3. We can see that the speedups are very significant, even when fusion can only be applied once (i.e., in $t1$). We also see that the more we can apply foldr/build-fusion, the more significant our speedup becomes.

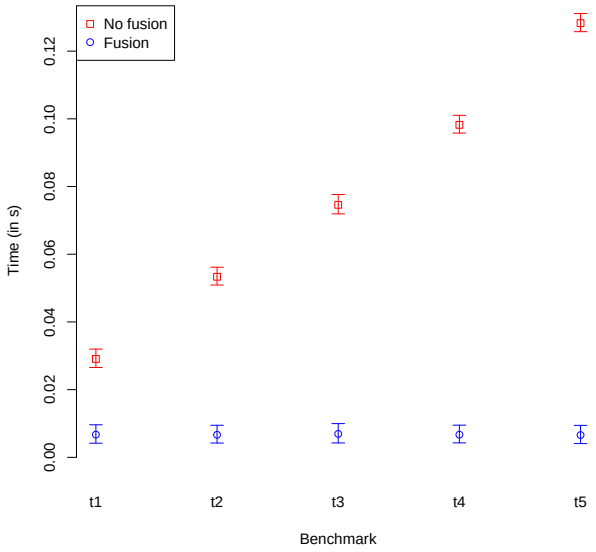


Fig. 3. Benchmarking the specified functions with and without fusion

VI. Related Work

A. Applications of Folds and Fusion

There is a long line of work on writing recursive functions in terms of structured recursion schemes, as well as proving fusion properties of these and exploiting them for deriving efficient programs.

Bird and Meertens [12, 13] have come up with several equational laws for recursion schemes to serve them in their work on program calculation.

With their Squiggol calculus, Meijer et al. [4] promote the use of structured recursion by means of recursion operators. These operators are defined in a datatype generic way and are equipped with a number of algebraic laws that enable equivalence-preserving program transformations.

Gibbons [14] promotes explicit programming in terms of folds and unfolds, which he calls *origami* programming. Unfolds are the dual of folds, and capture a special case of builds.

Gill et al. [3] present the foldr/build-fusion rule and discuss its benefits. They mention that it would be desirable, yet highly challenging, for the compiler to notice whether functions can be expressed in terms of *foldr* and *build*. That would allow programmers to write programs in whatever style they like.

Stream fusion [15] is an alternative to foldr/build-fusion. It has the benefits of easily being able to fuse zips and left folds. However, at the time of writing, there is no

known reliable method of optimising uses of $concatMap$. $concatMap$ is important because it represents the entire class of nested list computations, including list comprehensions [16].

B. Automatic Discovery

The *HLint* [11] tool is designed to recognize various code patterns and offer suggestions for improving them. In particular, it recognizes various forms of explicit recursion and suggests the use of appropriate higher-order functions like map, $filter$ and $foldr$ that capture these recursion patterns. As we already showed in Section V-A, we are able to detect more instances of folds for Haskell lists than HLint. Moreover, HLint makes no effort to detect folds for other algebraic datatypes.

Sittampalam and de Moor [17] present a semi-automatic approach to $foldr$ fusion based on the MAG system. In their approach, the programmer specifies the initial program, a specification of the target program and suitable rewrite rules. Then the MAG system will attempt to derive the target implementation by applying the rewrite rules.

VII. Conclusions

Programmers prefer high-level programming languages in order to be able to write elegant, maintainable code. However, performance remains important, i.e. the code needs to be able to be translated to efficient assembly. The conflict between those two goals can be solved fairly well by using advanced compilers and smart optimisations.

We added such a smart optimisation. It allows the programmer to write small, composable functions, either explicitly recursive or using a higher-order fold. More complex functions van be expressed as a combination of these small building blocks. When compiled naively, this would impose penalties on the performance, because of the overhead of allocating, building and destroying intermediate datastructures. By using foldr/build-fusion, we can avoid this overhead.

Our thesis has lifted the limitation that functions have to be written in a certain style in order to benefit from foldr/build-fusion, and extends it to work on any directly inductive datastructure instead of just lists.

Future work. A number of routes are still open for exploration. In particular, we mention mutually recursive functions, mutually recursive datatypes and GADTs [18]. Folds and builds exists for these types [19, 20], but our detection rules are incapable of finding these.

References

[1] John E. Gaffney, "Estimating the number of faults in code," *Software Engineering, IEEE Transactions on*, vol. SE-10, no. 4, pp. 459–464, 1984.

[2] Philip Wadler, "Deforestation: transforming programs to eliminate trees," *Theoretical Computer Science*, vol. 73, no. 2, pp. 231 – 248, 1990.

[3] Andrew Gill, John Launchbury, and Simon L. Peyton Jones, "A short cut to deforestation," in *Proceedings of the conference on Functional programming languages and computer architecture*, New York, NY, USA, 1993, FPCA '93, pp. 223–232, ACM.

[4] Erik Meijer, Maarten Fokkinga, and Ross Paterson, "Functional programming with bananas, lenses, envelopes and barbed wire," 1991, pp. 124–144, Springer-Verlag.

[5] Graham Hutton, "A tutorial on the universality and expressiveness of fold," *J. Funct. Program.*, vol. 9, no. 4, pp. 355–372, July 1999.

[6] The GHC Team, "The glasgow haskell compiler," 1989.

[7] The GHC Team, "Extending and using ghc as a library: Compiler plugins," 2011.

[8] Andrew Tolmach and Tim Chevalier, "An external representation for the ghc core language," 2009.

[9] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare, "Playing by the rules: rewriting as a practical optimisation technique in ghc," in *Haskell Workshop*, 2001, vol. 1, pp. 203–233.

[10] Tim Sheard and Simon Peyton Jones, "Template meta-programming for haskell," *SIGPLAN Not.*, vol. 37, no. 12, pp. 60–75, Dec. 2002.

[11] Neil Mitchell, *HLint Manual*, 2006.

[12] Richard Bird, "Constructive functional programming," in *Marktoberdorf International Summer school on Constructive Methods in Computer Science*, M. Broy, Ed. 1989, NATO Advanced Science Institute Series, Springer Verlag.

[13] Lambert Meertens, "Algorithmics – towards programming as a mathematical activity," in *CWI symposium on Mathematics and Computer Science*. 1986, pp. 289–334, North-Holland.

[14] Jeremy Gibbons, "Origami programming," in *The Fun of Programming*, Jeremy Gibbons and Oege de Moor, Eds., Cornerstones in Computing, pp. 41–60. Palgrave, 2003.

[15] Duncan Coutts, Roman Leshchinskiy, and Don Stewart, "Stream fusion. from lists to streams to nothing at all," in *ICFP'07*, 2007.

[16] Duncan Coutts, "Stream fusion: practical shortcut fusion for co-inductive sequence types," 2010.

[17] Ganesh Sittampalam and Oege de Moor, "Mechanising fusion," in *The Fun of Programming*, Jeremy Gibbons and Oege de Moor, Eds., Cornerstones in Computing, pp. 79–129. Palgrave, 2003.

[18] James Cheney and Ralf Hinze, "First-class phantom types," 2003.

[19] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring, "Generic programming with fixed points for mutually recursive datatypes," in *ACM Sigplan Notices*. ACM, 2009, vol. 44, pp. 233–244.

[20] Patricia Johann and Neil Ghani, "Foundations for structured programming with gadts," in *ACM SIGPLAN Notices*. ACM, 2008, vol. 43, pp. 297–308.

Inhoudsopgave

1 Inleiding

Laten we beginnen bij het begin. Al van bij de ontwikkeling van de eerste computers – naar onze hedendaagse normen vrij rudimentaire machines – was het noodzakelijk om in de gebruikte programmeertalen *controlestructuren* te voorzien. Deze instructies laten toe invloed uit te oefenen op de manier waarop het programma wordt uitgevoerd. In assembleertaal zijn dit de verschillende *sprong* instructies (`jmp`, `je`...). Typisch zal de instructie voor de spronginstructie een testinstructie zijn. Enkel gebruik maken van simpele tests en sprongen zonder duidelijke consistentie in de manier waarop deze gebruikt worden kan echter leiden tot zogenaamde ”spaghetti code”. Daarmee bedoelen we code die zowel moeilijk te lezen als te onderhouden is.

In latere programmeertalen (initieel talen zoals ALGOL, gevolgd door talen als C), maakte het concept *gestructureerd programmeren* een opmars. Dit betekent dat controlestructuren van een hoger abstractieniveau, zoals bijvoorbeeld `for`- en `while`-lussen, werden geïntroduceerd. Deze programmeertalen laten echter meestal wel nog toe om *expliciete* sprongen te maken door middel van de `goto` instructie[1]. Dit wordt geïllustreerd in Figuur 1.1.

De versie die gebruikt maakt van `for` is eenvoudiger leesbaar voor programmeurs die bekend zijn met dit concept. Het is immers niet langer nodig om de labels en `goto` instructies manueel te matchen en de relatie te bestuderen: het gebruikte keyword kondigt onmiddelijk de gebruikt controlestructuur aan (dit wordt meestal ook visueel ondersteund door gebruik te maken van indentatie).

Bovendien is het mogelijk formele eigenschappen uit te drukken over programma’s die geschreven zijn in deze stijl, bijvoorbeeld met Floyd-Hoare logica [10]. Dit leidde zelfs tot de conclusie dat het gebruik van `goto` volle-

[1] Merk op dat deze programmeertalen door een compiler worden omgezet naar machinetaal, waarin wel nog sprongen voorkomen. Dit vormt echter geen probleem voor leesbaarheid, sinds de meeste programmeurs deze machinetaal slechts zelden zullen bekijken.

```c
int sum_squares_for(int n) {
  int i, sum = 0;

  for(i = 1; i <= n; i++) {
    sum += i * i;
  }

  return sum;
}
```

```c
int sum_squares_goto(int n) {
  int i, sum = 0;

  i = 1;
  start: sum += i * i;
  i++;
  if(i <= n) goto start;

  return sum;
}
```

Figuur 1.1: Twee semantisch equivalente programma's in de programmeertaal C, links één met hoger-niveau controlestructuren en rechts één met `goto`*'s.*

dig vermeden moet worden [8].

Een soortgelijke redenering is te maken over *functionele programmeertalen*. Deze talen maken geen gebruik van `goto` instructies, maar implementeren controlestructuren door middel van *recursie*.

Deze programmeertalen bieden een hoog abstractieniveau en moedigen de programmeurs aan om gebruik te maken van *hogere-orde* functies (bv. *map*, *filter*, *any*, ...). Op deze manier is geen expliciete recursie nodig. Dit biedt verschillende voordelen:

1. Voor een programmeur die bekend is met de gebruikte hogere-orde functies is het mogelijk de code veel sneller te begrijpen [9]: men herkent onmiddelijk het patroon dat aangeboden wordt door de functie en dient enkel de argumenten van deze functie te bestuderen.

2. Door gebruik te maken van hogere-orde functies wordt de code beknopter. Eerder is aangetoond dat het aantal fouten in code proportioneel is tot de grootte van de codebase [11]. Hieruit kunnen we concluderen dat het gebruik van hogere-orde functies het aantal fouten in programma's zou moeten reduceren.

3. Het is mogelijk eigenschappen éénmaal te bewijzen over een hogere-orde functie voor willekeurige argumenten. Dit spaart ons werk uit als we willen redeneren over code, want de bewijzen gelden dan voor elke applicatie van deze hogere-orde functie.

4. Ook de compiler kan gebruik maken van deze eigenschappen, om verschillende optimalisaties uit te voeren op de code.

Deze redenen vormen een sterke motivatie om in deze talen geen expliciete

recursie te gebruiken als een hogere-orde functie beschikbaar is. Toch blijkt dat veel programmeurs nog gebruik maken van expliciete recursie.

Enkele redenen hiervoor zijn bijvoorbeeld dat de programmeur niet bekend is met de hogere-orde functie, of dat er geen tijd is om zijn zelfgeschreven functie te herschrijven op basis van bestaande hogere-orde functies. We zien zelfs dat we voorbeelden terugvinden van expliciete recursie in code geschreven door gevorderde gebruikers van functionele programmeertalen[2].

De hierboven beschreven voordelen vormen de basismotivatie voor het onderzoek dat we in deze thesis vericht hebben. We richten ons op functies die geschreven zijn in een expliciete recursieve stijl en onderzoeken in welke gevallen het mogelijk is deze automatisch om te zetten in functies die gebruik maken van de hogere-orde hulpfuncties. Op die manier kan de programmeur code schrijven in om het even welke stijl, en toch genieten van de verschillende optimalisaties.

We hanteren hiervoor de volgende concrete aanpak:

1. In hoofdstuk 3 tonen we aan hoe functies die expliciete recursie gebruiken maar wel een specifiek soort patroon (meer bepaald *catamorfismes* [23]) volgen kunnen gedetecteerd worden. Eveneens leggen we uit hoe deze door middel van herschrijfregels vertaald kunnen worden naar een versie die een hogere-orde fold functie gebruikt in plaats van expliciete recursie.

2. Tevens leggen we ook uit hoe we functies die geschreven kunnen worden als een toepassing van build kunnen detecteren en vertalen naar een versie die effectief gebruikt maakt van build. Dit wordt besproken in hoofdstuk 4. Merk op dat build op zich geen hogere-orde functie is, maar dat we zowel fold als build nodig hebben om *foldr/build-fusion* toe te passen, een bekende optimalisatie.

3. We implementeerden een GHC Compiler Plugin die deze detecties en vertalingen automatisch kan uitvoeren tijdens de compilatie van een Haskell-programma. Deze plugin werkt zowel voor de typische folds over lijsten in Haskell ($[a]$), maar ook voor andere (direct) recursieve datatypes, gedefinieerd door de gebruiker. In hoofdstuk 5 bespreken we de implementatie van deze plugin.

4. We onderzochten het aantal functies in enkele bekende Haskell packages die kunnen herschreven worden met behulp van een hogere orde

[2]Zo vinden we bijvoorbeeld zelfs veel voorbeelden van expliciete recursieve functies die kunnen geschreven worden met behulp een hogere-orde fold functie in de broncode van GHC (zie sectie 6.1).

fold-functie. Deze blijken in vele packages aanwezig te zijn. Ook bekijken we de resultaten van enkele benchmarks na automatische foldr/build-fusion. Omdat foldr/build-fusion de compiler toelaat om tussentijdse allocatie te vermijden, zien we hier zeer grote versnellingen. De resultaten hiervan zijn terug te vinden in hoofdstuk 6.

Dit heeft veel praktische toepassingen en om ons werk dan ook zo toegankelijk mogelijk te maken, werken we ook aan een Engelstalig artikel over hetzelfde onderwerp, “Bringing Functions into the Fold”. Dit artikel zullen we indienen voor de ACM SIGPLAN Haskell Symposium 2013 conferentie.

We kozen Haskell als programmeertaal voor ons onderzoek. In het volgende hoofdstuk zullen we kort ingaan op de eigenschappen van deze programmeertaal die we gebruiken in deze thesis.

2 Achtergrond

We kozen voor de pure functionele programmeertaal Haskell [22][19] omwille van verschillende redenen:

- Het Haskell Prelude[1] en de beschikbare bibliotheken bieden een waaier aan hogere-orde functies aan.
- Haskell is een sterk getypeerde programmeertaal. Na het initiële parsen en typechecken van de code is deze type-informatie is beschikbaar in elke stap van de compilatie. Deze types geven ons meer informatie die we kunnen gebruiken in de transformaties. Bovendien maakt Haskell gebruik van type inference [14], wat ervoor zorgt dat de programmeur meestal zelf geen types moet opgeven.
- De de-facto standaard Haskell Compiler, GHC [29], laat via een plugin-systeem toe om code te manipuleren op een relatief eenvoudige manier (zie sectie 5.2).

In dit hoofdstuk geven we een zeer beknopt overzicht van Haskell, en lichten we ook enkele relevante hogere-orde functies toe.

2.1 Haskell: types en functies

Haskell is gebaseerd op de lambda-calculus. Dit is een formeel, doch eenvoudig systeem, om aan logisch redeneren te doen. Het beschikt over een zeer beperkt aantal basisoperaties. We beginnen onze Haskell-introductie dan ook bij de lambda-calculus.

$e ::= x$ -- Een variabele
$\mid e\ e$ -- Functie-applicatie (links-associatief)

[1] Het Prelude is de module die impliciet in elk Haskell-programma wordt geïmporteerd. De functies hieruit zijn dus rechtstreeks te gebruiken zonder dat men een bibliotheek moet importeren.

$| \lambda x \to e$ -- Functie-abstractie (rechts-associatief)

Dit is natuurlijk een zeer beperkte syntax en Haskell breidt deze sterk uit. Beschouw bijvoorbeeld de volgende Haskell-functie (links) en het lambda-calculus equivalent (rechts):

$$middle\ x\ y = (x + y)\ /\ 2 \qquad\qquad middle = \lambda x \to \lambda y \to (/)\ ((+)\ x\ y)\ 2$$

In tegenstelling tot de lambda-calculus [3], is Haskell ook een *sterk getypeerde* programmeertaal. Functies worden, naast een definitie, ook voorzien van een *type-signatuur*:

$$middle :: Float \to Float \to Float$$

Net zoals lambda-abstracties is de $\to$ (een operator op type-niveau) in type-signaturen rechts-associatief. Deze type-signatuur is dus equivalent aan $Float \to (Float \to Float)$. Dit concept heet *currying*: we kunnen *middle* beschouwen als een functie die één Float argument neemt en functie van het type $Float \to Float$ als resultaatwaarde heeft, of als een functie die twee *Float*-argumenten neemt en een *Float* als resultaatwaarde heeft.

In de lambda-calculus is het niet mogelijk om direct recursieve functies te schrijven. Er bestaat echter een elegante oplossing: de *fixpoint-combinator*. Een fixpoint-combinator is een functie g waarvoor geldt dat voor elke functie f: $g\ f = f\ (g\ f)$. Hierdoor is het mogelijk een functie door te geven voor een recursieve oproep, zonder deze functie expliciet een naam te geven.

Eén van de de bekendste voorbeelden hiervan is de *Y-combinator*.

Definitie 2.1.1.

$$Y \quad \equiv \quad \lambda f \to (\lambda x \to f\ (x\ x))\ (\lambda x \to f\ (x\ x))$$

Stelling. *Y is een fixpoint-combinator, dus:*

$$Y\ f \equiv f\ (Y\ f)$$

Bewijs.

$$\begin{array}{ll} & Y\ f \\ \equiv & \{\ \text{def}\ Y\ \} \end{array}$$

$$
\begin{aligned}
& (\lambda f \to (\lambda x \to f\,(x\,x))\ (\lambda x \to f\,(x\,x)))\ f \\
\equiv\ & \quad \{\ \beta\text{-reductie}\ \} \\
& (\lambda x \to f\,(x\,x))\ (\lambda x \to f\,(x\,x))) \\
\equiv\ & \quad \{\ \beta\text{-reductie}\ \} \\
& f\,((\lambda x \to f\,(x\,x))\ (\lambda x \to f\,(x\,x))) \\
\equiv\ & \quad \{\ \lambda\text{-abstractie}\ \} \\
& f\,((\lambda f \to (\lambda x \to f\,(x\,x))\ (\lambda x \to f\,(x\,x)))\ f) \\
\equiv\ & \quad \{\ \text{def}\ Y\ \} \\
& f\,(Y\,f)
\end{aligned}
$$

□

In tegenstelling tot de lambda-calculus is het in Haskell niet nodig om gebruik te maken van fixpoint-combinators: men kan eenvoudig expliciet recursieve definities geven door de functie zelf bij naam aan te roepen.

$$
\begin{aligned}
& \mathit{fib} :: \mathit{Int} \to \mathit{Int} \\
& \mathit{fib}\ n = \textbf{if}\ n \leqslant 1\ \textbf{then}\ 1\ \textbf{else}\ \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2)
\end{aligned}
$$

Verder breidt Haskell de lambda-calculus uit met *algebraïsche datatypes* en *pattern matching*. Een algebraïsch datatype is in Haskell typisch een *somtype* (keuze tussen verschillende types) van *producttypes* (combinatie van verschillende types).

Beschouw het volgende voorbeeld:

$$
\begin{aligned}
& \textbf{data}\ \mathit{Topping} = \mathit{Salami} \mid \mathit{Mozarella} \mid \mathit{Peppers} \\
& \textbf{data}\ \mathit{Pizza} \\
& \quad = \mathit{Plain} \\
& \quad \mid \mathit{ExtraToppings}\ \mathit{Topping}\ \mathit{Topping}
\end{aligned}
$$

Topping is een somtype met drie verschillende constructoren zonder meer informatie. *Pizza* is ook een somtype van twee verschillende constructoren, waarvan één een producttype is van twee *Topping*-types.

Met behulp van pattern-matching kan de onderliggende constructor onderzocht worden:

$$
\begin{aligned}
& \mathit{toppingPrice} :: \mathit{Topping} \to \mathit{Double} \\
& \mathit{toppingPrice}\ \mathit{Salami} \quad = 0.50 \\
& \mathit{toppingPrice}\ \mathit{Mozarella} = 0.50 \\
& \mathit{toppingPrice}\ \mathit{Peppers} \quad = 0.30
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{pizzaPrice} :: \mathit{Pizza} \to \mathit{Double} \\
&\mathit{pizzaPrice}\ \mathit{Plain} &&= 5.20 \\
&\mathit{pizzaPrice}\ (\mathit{ExtraToppings}\ \mathit{t1}\ \mathit{t2}) &&= \\
&\quad 5.40 + \mathit{toppingPrice}\ \mathit{t1} + \mathit{toppingPrice}\ \mathit{t2}
\end{aligned}
$$

Het zou ons te ver voeren om het volledig typesysteem van Haskell hier te bespreken. We beperken ons dan ook tot de kenmerken die we nodig hebben in deze scriptie. Een belangrijk en bijzonder interessant kenmerk van het typesysteem dat we doorheen deze thesis gebruiken is *type-polymorfisme*. Dit laat ons toe om met éénzelfde set functies en datatypes te werken met waarden van verschillende types. Beschouw bijvoorbeeld de identiteitsfunctie *id*. Deze functie kan op een waarde van om het even welk type toegepast worden en is bijgevolg polymorf.

$$
\begin{aligned}
&\mathit{id} :: a \to a \\
&\mathit{id}\ x = x
\end{aligned}
$$

Haskell-functies kunnen ook andere functies als argumenten nemen. Deze functies worden hogere-orde functies genoemd en zijn alomtegenwoordig in Haskell-code.

Een zeer bekend voorbeeld van een hogere-orde functie is functiecompositie, $(\circ)$.

$$
\begin{aligned}
&(\circ) :: (b \to c) \to (a \to b) \to (a \to c) \\
&f \circ g = \lambda x \to f\ (g\ x)
\end{aligned}
$$

Dit concept komt uit de wiskunde, waar eveneens notatie $f \circ g$ gebruikt wordt. Deze functie laat ons ook toe om zogenaamde pijplijn-code te schrijven, waarbij we de resultaatwaarde van één functie telkens gebruiken als argument van een volgende functie:

$$f \circ g \circ \ldots \circ h$$

In sectie 6.4 zien we dat ons werk de compiler toelaat bepaalde instanties van dergelijke pijplijn-code te optimaliseren.

Een andere veelgebruikte hogere-orde functie is $(\$)$. Deze functie staat voor *functie-applicatie*.

$$
\begin{aligned}
&(\$) :: (a \to b) \to a \to b \\
&f \$ x = f\ x
\end{aligned}
$$

Dit lijkt misschien een nutteloze functie: waarom zou men $f\,\$\,x$ schrijven als men evengoed $f\ x$ kan schrijven? Het grote voordeel van het gebruik van ($\$$) ligt echter in de rechtse associativiteit van deze operator. Men kan ($\$$) dus als het ware gebruiken om haakjes te vervangen. Dit kan in sommige gevallen de code leesbaarder maken:

$$\begin{array}{ll} & f\ (g\ (h\ (i\ (j\ x)))) \\ \equiv & \quad \{\ \text{def}\ \$\ \} \\ & f\,\$\,g\,\$\,h\,\$\,i\,\$\,j\ x \end{array}$$

Behalve polymorfe functies bestaan er ook polymorfe datatypes. Een veelgebruikt voorbeeld hiervan is de *tuple*, dat een paar van waarden voorstelt.

$$\textbf{data}\ (a, b) = (a, b)$$

Het polymorfisme van dit datatype laat ons toe om waarden van eender welk type te koppelen, bijvoorbeeld een *String* en een *Int*:

$$\begin{array}{l} \mathit{jasper} :: (\mathit{String}, \mathit{Int}) \\ \mathit{jasper} = (\texttt{"Jasper"}, 22) \end{array}$$

Meerder waarden kunnen ook gekoppeld worden door deze tuples te *nesten*[2]. Zo krijgen we bijvoorbeeld:

$$\begin{array}{l} \mathit{zeroes} :: (((\mathit{Int}, \mathit{Integer}), \mathit{Double}), \mathit{Float}) \\ \mathit{zeroes} = (((0, 0), 0), 0) \end{array}$$

Naast polymorf kunnen datatypes ook recursief zijn. Het de-facto voorbeeld hiervan is de lijst.

$$\begin{array}{l} \textbf{data}\ [a] \\ \quad = a : [a] \\ \quad \mid\ [\,] \end{array}$$

Lijsten zijn alomtegenwoordig in functionele programma's en verdienen speciale aandacht. Zo worden bijvoorbeeld *String*s in Haskell ook voorgesteld als karakter-lijsten:

[2] Eveneens kunnen we meerde waarden koppelen door triples, of andere ... *n*-tuples te gebruiken. Op die manier krijgen we:

$$\begin{array}{l} \mathit{zeroes} :: (\mathit{Int}, \mathit{Integer}, \mathit{Double}, \mathit{Float}) \\ \mathit{zeroes} = (0, 0, 0, 0) \end{array}$$

Deze *n*-tuples worden echter niet gebruik in onze thesis.

$$\textbf{type}\ String = [Char]$$

Omdat een lijst een recursief datatype is, is het zeer gebruikelijk om recursieve functies over lijsten te schrijven. Met behulp van de standaardfunctie $toUpper :: Char \to Char$ kan men bijvoorbeeld een functie schrijven die een volledige *String* omzet naar drukletters:

$$\begin{aligned} &upper :: String \to String \\ &upper\ [\,] \quad\ = [\,] \\ &upper\ (x : xs) = toUpper\ x : upper\ xs \end{aligned}$$

2.2 Higher-order functies

Omdat recursieve functies over lijsten alomtegenwoordig zijn, is het vaak mogelijk patronen te onderscheiden. Beschouw even de volgende twee functies, *sum* en *product*.

$$\begin{aligned} &sum :: [Int] \to Int \\ &sum\ [\,] \quad\ = 0 \\ &sum\ (x : xs) = x + sum\ xs \\ &product :: [Int] \to Int \\ &product\ [\,] \quad\ = 1 \\ &product\ (x : xs) = x * product\ xs \end{aligned}$$

Deze patronen kunnen geïmplementeerd worden door middel van hogereorde functies. In het bovenstaande voorbeeld zijn de functies eenvoudige voorbeelden van het *foldr* patroon.

$$\begin{aligned} &foldr :: (a \to b \to b) \to b \to [a] \to b \\ &foldr\ \_\ z\ [\,] \quad\ = z \\ &foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs) \end{aligned}$$

Als we *sum* en *product* herschrijven op basis van *foldr*, krijgen we veel beknoptere definities, die semantisch equivalent zijn aan de expliciet recursieve versies. Deze zijn bovendien sneller te lezen door ervaren programmeurs:

$$\begin{aligned} &sum' :: [Int] \to Int \\ &sum' = foldr\ (+)\ 0 \\ &product' :: [Int] \to Int \\ &product' = foldr\ (*)\ 1 \end{aligned}$$

Andere voorbeelden van hogere-orde functies zijn *map* en *filter*. *map* laat ons toe een functie uit te voeren op elk element van een lijst:

$$
\begin{aligned}
&map :: (a \to b) \to [a] \to [b] \\
&map\ f = foldr\ (\lambda x\ xs \to f\ x : xs)\ [\,] \\
&upper' :: String \to String \\
&upper' = map\ toUpper
\end{aligned}
$$

De *filter*-functie wordt gebruikt om bepaalde elementen uit een lijst te selecteren:

$$
\begin{aligned}
&filter :: (a \to Bool) \to [a] \to [a] \\
&filter\ f = foldr\ (\lambda x\ xs \to \textbf{if}\ f\ x\ \textbf{then}\ x : xs\ \textbf{else}\ xs)\ [\,] \\
&odds :: [Int] \to [Int] \\
&odds = filter\ odd
\end{aligned}
$$

2.3 De universele eigenschap van fold

Het feit dat we zowel *map* als *filter* schrijven met behulp van *foldr* duidt aan dat *foldr* een zeer interessante functie is. Meer bepaald, de universele eigenschap van fold [16] is weergegeven in stelling 2.3.1.

Stelling 2.3.1.

$$
g = foldr\ f\ v \qquad \Leftrightarrow \qquad \begin{aligned} g\ [\,] &= v \\ g\ (x : xs) &= f\ x\ (g\ xs) \end{aligned}
$$

Concreet betekent dit dat we een functie g kunnen herschrijven in termen van *foldr* zodra we een f en v vinden die aan de bovenstaande voorwaarden voldoen.

Ook betekent dit dat er slechts één *foldr* is voor een lijst – elke alternatieve definitie is hieraan isomorf [16]. Er is dus een wederzijds verband tussen het type $[a]$ en de functie *foldr*. De vraag naar het bestaan van een bijectie tussen algebraïsche datatypes en fold-functies dringt zich dus op.

Deze vraag kan affirmatief beantwoord worden: een dergelijke bijectie bestaat, ze legt bovendien het verband tussen een datatype en het overeenkomstige *catamorfisme*: de unieke manier om een algebraïsch datatype stap

voor stap te reduceren tot één enkele waarde. We kunnen deze catamorfismes eenvoudig afleiden voor algebraïsche datatypes.

Om dit beter te verstaan, hebben we het concept van een *algebra* nodig. Wanneer we een catamorfisme toepassen op een datatype, interpreteren we dit datatype in een bepaalde algebra, door elke constructor te vervangen door een operator uit deze algebra. Zo is *sum* als het ware een interpretatie in de som-algebra, die $(:)$ en $[]$ vervangt door respectievelijk $+$ en 0:

$$\begin{aligned} &\mathit{foldr}\ (+)\ 0\ (1\ :\ (2\ :\ (3\ :\ (4\ :\ [])))) \\ \equiv\ &\qquad (1 + (2 + (3 + (4 + 0)))) \end{aligned}$$

Dit idee laat ons toe folds te definiëren voor andere datatypes. Beschouw bijvoorbeeld een eenvoudig boom-type:

```
data Tree a
  = Leaf a
  | Branch (Tree a) (Tree a)
```

Door een functie-argument te specificeren voor elke constructor, kunnen we nu een fold definiëren voor het type *Tree*:

```
foldTree :: (a → b)          -- Operator voor leaf
         → (b → b → b)       -- Operator voor branch
         → Tree a            -- Input tree
         → b                 -- Resultaat van de fold
foldTree leaf _      (Leaf x)      = leaf x
foldTree leaf branch (Branch x y) =
  branch (foldTree leaf branch x) (foldTree leaf branch y)
```

En met behulp van deze functie kunnen we dus eenvoudig recursieve functies over bomen schrijven. *sumTree*, bijvoorbeeld, berekent de som van de waarden van de bladeren van een boom:

```
sumTree :: Tree Int → Int
sumTree = foldTree id (+)
```

We concluderen dat een fold voor een bepaald algebraïsch datatype dus eenvoudig is af te leiden uit de definitie van dat datatype. Bijgevolg kunnen we dit ook automatisch doen.

Template Haskell [27] is een Haskell-extensie die toelaat om aan type-safe compile-time meta-programmeren te doen. Op deze manier kunnen we Haskell manipuleren met Haskell.

We implementeerden een algoritme in Template Haskell om de fold horende bij een datatype automatisch te genereren. Zo kan bijvoorbeeld *foldTree* gegenereerd worden door:

$$\$(deriveFold\ ``Tree\ \texttt{"foldTree"})$$

Het algoritme werkt als volgt. We gebruiken de types $Tree\ a$ en $[a]$ als voorbeelden.

1. De fold neemt als laatste argument altijd een waarde van het opgegeven type, en geeft een waarde van het type b terug.

   $$foldTree :: ... \to Tree\ a \to b$$
   $$foldList :: ... \to [a] \to b$$

2. Per constructor wordt er een extra argument meegeven.

   $$foldTree :: <LeafArg> \to <BranchArg> \to Tree\ a \to b$$
   $$foldList :: <ConsArg> \to <NilArg> \to [a] \to b$$

3. Wat zijn nu de concrete types van deze argumenten? Laten we eerst de types van de constructoren beschouwen:

   $$Leaf :: a \to Tree\ a$$
   $$Branch :: Tree\ a \to Tree\ a \to Tree\ a$$
   $$(:) :: a \to [a] \to [a]$$
   $$[] :: [a]$$

4. Deze constructoren geven de subtermen aan en corresponderen dus met de verschillende argumenten. De recursie wordt echter afgehandeld door de fold functie, en dus is elke recursieve subterm al gereduceerd tot een waarde van het type b. Eveneens is b het type van het resultaat. We vinden:

   $$<LeafArg> = a \to b$$
   $$<BranchArg> = b \to b \to b$$
   $$<ConsArg> = a \to b \to b$$
   $$<NilArg> = b$$

   En dus:

   $$foldTree :: (a \to b) \to (b \to b \to b) \to Tree\ a \to b$$
   $$foldList :: (a \to b \to b) \to b \to [a] \to b$$

5. Eenmaal de type-signaturen bepaald zijn is het genereren van de implementatie redelijk eenvoudig. Elke functieparameter krijgt een naam naar de constructor. Vervolgens genereren we een *go* functie. Dit is een toepassing van de Static Argument Transformation [6].

$$
\begin{aligned}
&\mathit{foldTree} :: (a \to b) \to (b \to b \to b) \to \mathit{Tree}\ a \to b \\
&\mathit{foldTree}\ \mathit{leaf}\ \mathit{branch} = \mathit{go} \\
&\quad \textbf{where} \\
&\qquad \mathit{go}\ (\mathit{Leaf}\ x) \quad\ = \mathit{leaf}\ x \\
&\qquad \mathit{go}\ (\mathit{Branch}\ x\ y) = \mathit{branch}\ (\mathit{go}\ x)\ (\mathit{go}\ y) \\
&\mathit{foldList} :: (a \to b \to b) \to b \to [a] \to b \\
&\mathit{foldList}\ \mathit{cons}\ \mathit{nil} = \mathit{go} \\
&\quad \textbf{where} \\
&\qquad \mathit{go}\ (x : y) = \mathit{cons}\ x\ (\mathit{go}\ y) \\
&\qquad \mathit{go}\ [\,] \quad\ = \mathit{nil}
\end{aligned}
$$

   De *go* functie inspecteert simpelweg de constructor en roept dan het corresponderende functie-argument met als argumenten de gereduceerde subtermen. Een gereduceerde niet-recursieve subterm t is gewoon die subterm t, en voor een recursieve subterm is dit $\mathit{go}\ t$.

2.4 Fusion: Folds en Builds

2.4.1 Wat is fusion?

Naast de verschillende voordelen op vlak van *refactoring*, is het ook mogelijk *optimalisaties* door te voeren op basis van deze hogere-orde functions.

Beschouw de volgende twee versies van een functie die de som van de kwadraten van de oneven nummers in een lijst berekent:

$$
\begin{aligned}
&\mathit{sumOfSquaredOdds} :: [\mathit{Int}] \to \mathit{Int} \\
&\mathit{sumOfSquaredOdds}\ [\,] = 0 \\
&\mathit{sumOfSquaredOdds}\ (x : xs) \\
&\quad \mid \mathit{odd}\ x \qquad\qquad\ = x \uparrow 2 + \mathit{sumOfSquaredOdds}\ xs \\
&\quad \mid \mathit{otherwise} \qquad\ = \mathit{sumOfSquaredOdds}\ xs \\
&\\
&\mathit{sumOfSquaredOdds}' :: [\mathit{Int}] \to \mathit{Int} \\
&\mathit{sumOfSquaredOdds}' = \mathit{sum} \circ \mathit{map}\ (\uparrow 2) \circ \mathit{filter}\ \mathit{odd}
\end{aligned}
$$

Ervaren Haskell-programmeurs zullen steevast de tweede versie boven de eerste verkiezen. Het feit dat de tweede versie is opgebouwd uit kleinere, makkelijk te begrijpen functies maakt deze veel leesbaarder.

De eerste versie is echter efficiënter: deze berekent rechtstreeks het resultaat (een *Int*), terwijl de tweede versie twee tijdelijke $[Int]$ lijsten aanmaakt: een eerste als resultaat van *filter odd*, en een tweede als resultaat van *map* $(\uparrow 2)$.

In de ideale situatie willen we dus de efficiëntie van de eerste versie combineren met de leesbaarheid van de tweede versie. Dit wordt mogelijk gemaakt door *fusion* [34] [13].

We kunnen fusion best uitleggen aan de hand van een eenvoudig voorbeeld: *map/map-fusion*. Dit is een transformatie die gegeven wordt door stelling 2.4.1.

Stelling 2.4.1.

$$map\ f \circ map\ g \quad \equiv \quad map\ (f \circ g)$$

Bewijs. Deze equivalentie is eenvoudig te bewijzen via inductie. We bewijzen dit eerst voor de lege lijst $[\,]$. Voor $map\ f \circ map\ g$ krijgen we:

$$\begin{aligned}
&map\ f\ (map\ g\ [\,]) \\
\equiv\ &\quad \{\ \text{def } map\ [\,]\ \} \\
&map\ f\ [\,] \\
\equiv\ &\quad \{\ \text{def } map\ [\,]\ \} \\
&[\,] \\
\equiv\ &\quad \{\ \text{def } map\ [\,]\ \} \\
&map\ (f \circ g)\ [\,]
\end{aligned}$$

We nemen nu aan dat map/map-fusion correct is voor een willekeurige lijst xs en bewijzen dat de correctheid dan ook geldt voor een lijst $x : xs$.

$$\begin{aligned}
&map\ f\ (map\ g\ (x : xs)) \\
\equiv\ &\quad \{\ \text{def } map\ (:)\ \} \\
&map\ f\ (g\ x : map\ g\ xs) \\
\equiv\ &\quad \{\ \text{def } map\ (:)\ \} \\
&f\ (g\ x) : map\ f\ (map\ g\ xs) \\
\equiv\ &\quad \{\ \text{inductiehypothese}\ \} \\
&f\ (g\ x) : map\ (f \circ g)\ xs \\
\equiv\ &\quad \{\ \text{def } map\ (:)\ \} \\
&map\ (f \circ g)\ (x : xs)
\end{aligned}$$

□

GHC beschikt over een mechanisme om dit soort transformaties uit te voeren tijdens de compilatie, door middel van `{-# RULES -#}` pragma's [20]. Zo kunnen we bijvoorbeeld map/map-fusion implementeren door eenvoudigweg het volgende pragma te vermelden:

```
{-# RULES "map/map-fusion" forall f g xs.
    map f (map g xs) = map (f . g) xs #-}
```

Het nadeel van deze aanpak is echter dat het aantal vereiste rules kwadratisch stijgt in proportie tot het aantal hogere-orde functies dat op het datatype werkt – in dit geval lijsten.

Ter illustratie, als we bijvoorbeeld enkel de twee functies *map* en *filter* beschouwen, hebben we al vier rules nodig, en een bijkomende hulpfunctie *mapFilter*:

$$
\begin{aligned}
&\mathit{map}\ f \circ \mathit{map}\ g \equiv \mathit{map}\ (f \circ g)\\
&\mathit{map}\ f \circ \mathit{filter}\ g \equiv \mathit{mapFilter}\ f\ g\\
&\mathit{filter}\ f \circ \mathit{map}\ g \equiv \mathit{filter}\ (f \circ g)\\
&\mathit{filter}\ f \circ \mathit{filter}\ g \equiv \mathit{filter}\ (\lambda x \to f\ x \wedge g\ x)\\
&\mathit{mapFilter} :: (a \to b) \to (a \to \mathit{Bool}) \to [a] \to [b]\\
&\mathit{mapFilter}\ \_\ \_\ [\,] = [\,]\\
&\mathit{mapFilter}\ f\ g\ (x : xs)\\
&\quad \mid g\ x \qquad\quad = f\ x : \mathit{mapFilter}\ f\ g\ xs\\
&\quad \mid \mathit{otherwise} \;= \mathit{mapFilter}\ f\ g\ xs
\end{aligned}
$$

Maar als we nu een langere expressie $\mathit{map}\ f \circ \mathit{map}\ g \circ \mathit{filter}\ h$ hebben, krijgen we iets van de vorm $\mathit{map}\ f \circ \mathit{mapFilter}\ g\ h$, en dienen we weer nieuwe fusion-regels toe te voegen om deze expressie te kunnen fuseren. Het aantal nodige regels stijgt dus zeer snel.

Voor sommige modules ligt het aantal hogere-orde functies erg hoog, dus wordt deze aanpak onhaalbaar.

2.4.2 Foldr/build-fusion

Dit probleem wordt opgelost met *foldr/build-fusion*. We kunnen *foldr* beschouwen als een algemene manier om lijsten te *consumeren*. Hiervan is *build* de tegenhanger: een algemene manier om lijsten te *produceren*.

$$
\begin{aligned}
&\mathit{build} :: (\forall b.(a \to b \to b) \to b \to b) \to [a]\\
&\mathit{build}\ g = g\ (:)\ [\,]
\end{aligned}
$$

We kunnen nu bijvoorbeeld *map* en *filter* met behulp van *build* definiëren:

$$
\begin{aligned}
&\mathit{map} :: (a \to b) \to [a] \to [b] \\
&\mathit{map}\ f\ \mathit{ls} = \mathit{build}\ \$\ \lambda \mathit{cons}\ \mathit{nil} \to \\
&\quad \mathit{foldr}\ (\lambda x\ \mathit{xs} \to \mathit{cons}\ (f\ x)\ \mathit{xs})\ \mathit{nil}\ \mathit{ls} \\
&\mathit{filter} :: (a \to \mathit{Bool}) \to [a] \to [a] \\
&\mathit{filter}\ f\ \mathit{ls} = \mathit{build}\ \$\ \lambda \mathit{cons}\ \mathit{nil} \to \\
&\quad \mathit{foldr}\ (\lambda x\ \mathit{xs} \to \textbf{if}\ f\ x\ \textbf{then}\ \mathit{cons}\ x\ \mathit{xs}\ \textbf{else}\ \mathit{xs})\ \mathit{nil}\ \mathit{ls}
\end{aligned}
$$

Het nut van *build* wordt nu duidelijk: we gebruiken deze functie om te *abstraheren* over de concrete constructoren: in plaats van $(:)$ en $[\,]$ gebruiken we nu de abstracte *cons* en *nil* parameters.

De type-signatuur van *build* met het expliciet universeel gekwantificeerde type b is cruciaal. Stel dat dit niet het geval zou zijn, en dat we *build* zouden definiëren met de meest algemene type-signatuur:

$$
\begin{aligned}
&\mathit{build}' :: ((a \to [a] \to [a]) \to [a] \to t) \to t \\
&\mathit{build}'\ g = g\ (:)\ [\,]
\end{aligned}
$$

Dan zou code als *list123* well-typed zijn:

$$
\begin{aligned}
&\mathit{list123} :: [\mathit{Int}] \\
&\mathit{list123} = \mathit{build}'\ \$\ \lambda \mathit{cons}\ \mathit{nil} \to 1 : \mathit{cons}\ 2\ (\mathit{cons}\ 3\ [\,])
\end{aligned}
$$

We krijgen een lijst die zowel gebruikt maakt van de concrete constructoren als de abstracte versies. Dit leidt tot problemen: intuïtief laten de abstracte versies ons toe om de constructoren $(:)$ en $[\,]$ te *vervangen* door andere functies – en zoals we in Sectie 2.3 zagen, kunnen we het toepassen van *foldr* net beschouwen als het vervangen van de constructoren door de argumenten van *foldr*!

Als we echter ook nog letterlijk verwijzen naar $(:)$ en $[\,]$, is deze vervanging onmogelijk. Het universeel gekwantificeerde type b lost dit probleem op. De programmeur is verplicht een g mee te geven die werkt voor *elke* b, en hij weet niet welk type uiteindelijk geconstrueerd zal worden. Bijgevolg kan hij dus ook geen concrete constructoren gebruiken.

Nu we vastgesteld hebben dat enkel de abstracte versies van de constructoren gebruikt worden, laat dit idee ons toe om de productie en consummatie van een lijst te fuseren, zodanig dat er geen tijdelijke lijst moet worden aangemaakt. We werken dit nu formeel uit.

Stelling 2.4.1. *Als*

$$g :: \forall b.(A \to b \to b) \to b \to b$$

dan

$$\mathit{foldr}\ \mathit{cons}\ \mathit{nil}\ (\mathit{build}\ g) \quad \equiv \quad g\ \mathit{cons}\ \mathit{nil}$$

Bewijs. Van het type van een polymorfe functie kan een *gratis theorie* afgeleid worden [33]. Zo krijgen we voor g dat voor alle f_1, f_2 en h met als types:

$$\begin{aligned} h &:: B_1 \to B_2 \\ f_1 &:: A \to B_1 \to B_1 \\ f_2 &:: A \to B_2 \to B_2 \end{aligned}$$

de volgende implicatie geldt:

$$\begin{aligned} (\forall x\ xs_1\ xs_2.h\ xs_1 \equiv xs_2 \Rightarrow h\ (f_1\ x\ xs_1) \equiv f_2\ x\ xs_2) \Rightarrow \\ (\forall xs_1\ xs_2.h\ xs_1 \equiv xs_2 \Rightarrow h\ (g\ f_1\ xs_1) \equiv g\ f_2\ xs_2) \end{aligned}$$

De gelijkheid $h\ xs_1 \equiv xs_2$ kunnen we tweemaal substitueren, waardoor de implicatie herleid wordt tot:

$$\begin{aligned} (\forall x\ xs_2.h\ (f_1\ x\ xs_1) \equiv f_2\ x\ (h\ xs_1)) \Rightarrow \\ (\forall xs_2.h\ (g\ f_1\ xs_1) \equiv g\ f_2\ (h\ xs_1)) \end{aligned}$$

We kunnen deze implicatie nu instantiëren met: $f_1 := (:)$, $f_2 := \mathit{cons}$, en $h := \mathit{foldr}\ \mathit{cons}\ \mathit{nil}$. We krijgen dus:

$$\begin{aligned} (\forall x\ xs_2.\mathit{foldr}\ \mathit{cons}\ \mathit{nil}\ (x : xs_1) \equiv \mathit{cons}\ x\ (\mathit{foldr}\ \mathit{cons}\ \mathit{nil}\ xs_1)) \Rightarrow \\ (\forall xs_2.\mathit{foldr}\ \mathit{cons}\ \mathit{nil}\ (g\ (:)\ xs_1) \equiv g\ \mathit{cons}\ (\mathit{foldr}\ \mathit{cons}\ \mathit{nil}\ xs_1)) \end{aligned}$$

De linkerkant van de implicatie is triviaal geldig: dit is gewoon de definitie van *foldr* voor een niet-ledige lijst. Hieruit volgt dat:

$$(\forall xs_2.\mathit{foldr}\ \mathit{cons}\ \mathit{nil}\ (g\ (:)\ xs_2) \equiv g\ \mathit{cons}\ (\mathit{foldr}\ \mathit{cons}\ \mathit{nil}\ xs_2))$$

Deze gelijkheid kunnen we opnieuw instantiëren, ditmaal met $xs_2 := []$. Zo krijgen we:

$$
\begin{aligned}
& \mathit{foldr}\ \mathit{cons}\ \mathit{nil}\ (g\ (:)\ [\,]) \equiv g\ \mathit{cons}\ (\mathit{foldr}\ \mathit{cons}\ \mathit{nil}\ [\,]) \\
\equiv\ & \quad \{\ \text{def}\ \mathit{foldr}\ [\,]\ \} \\
& \mathit{foldr}\ \mathit{cons}\ \mathit{nil}\ (g\ (:)\ [\,]) \equiv g\ \mathit{cons}\ \mathit{nil} \\
\equiv\ & \quad \{\ \text{def}\ \mathit{build}\ \} \\
& \mathit{foldr}\ \mathit{cons}\ \mathit{nil}\ (\mathit{build}\ g) \equiv g\ \mathit{cons}\ \mathit{nil}
\end{aligned}
$$

□

Ter illustratie tonen we nu hoe met deze enkele fusion-regel onze elegantere versie van *sumOfSquaredOdds′* automatisch door GHC kan worden omgezet naar een efficiënte versie.

$$
\begin{aligned}
& \mathit{sumOfSquaredOdds'} \\
\equiv\ & \quad \{\ \text{inline}\ \mathit{sumOfSquaredOdds'}\ \} \\
& \mathit{sum} \circ \mathit{map}\ (\uparrow 2) \circ \mathit{filter}\ \mathit{odd} \\
\equiv\ & \quad \{\ \text{inline}\ \circ\ \} \\
& \lambda \mathit{ls} \to \mathit{sum}\ (\mathit{map}\ (\uparrow 2)\ (\mathit{filter}\ \mathit{odd}\ \mathit{ls})) \\
\equiv\ & \quad \{\ \text{inline}\ \mathit{filter}\ \} \\
& \lambda \mathit{ls} \to \mathit{sum}\ (\mathit{map}\ (\uparrow 2) \\
& \quad (\mathit{build}\ \$\ \lambda \mathit{cons}\ \mathit{nil} \to \\
& \quad\ \ \mathit{foldr}\ (\lambda x\ \mathit{xs} \to \textbf{if}\ \mathit{odd}\ x\ \textbf{then}\ \mathit{cons}\ x\ \mathit{xs}\ \textbf{else}\ \mathit{xs})\ \mathit{nil}\ \mathit{ls})) \\
\equiv\ & \quad \{\ \text{inline}\ \mathit{map}\ \} \\
& \lambda \mathit{ls} \to \mathit{sum} \\
& \quad (\mathit{build}\ \$\ \lambda \mathit{cons'}\ \mathit{nil'} \to \\
& \quad\ \ \mathit{foldr}\ (\lambda x\ \mathit{xs} \to \mathit{cons'}\ (x \uparrow 2)\ \mathit{xs})\ \mathit{nil'} \\
& \quad\ \ \quad (\mathit{build}\ \$\ \lambda \mathit{cons}\ \mathit{nil} \to \\
& \quad\ \ \quad\ \ \mathit{foldr}\ (\lambda x\ \mathit{xs} \to \textbf{if}\ \mathit{odd}\ x\ \textbf{then}\ \mathit{cons}\ x\ \mathit{xs}\ \textbf{else}\ \mathit{xs})\ \mathit{nil}\ \mathit{ls})) \\
\equiv\ & \quad \{\ \text{foldr/build-fusion}\ \} \\
& \lambda \mathit{ls} \to \mathit{sum} \\
& \quad (\mathit{build}\ \$\ \lambda \mathit{cons'}\ \mathit{nil'} \to \\
& \quad\ \ (\lambda \mathit{cons}\ \mathit{nil} \to \\
& \quad\ \ \quad \mathit{foldr}\ (\lambda x\ \mathit{xs} \to \textbf{if}\ \mathit{odd}\ x\ \textbf{then}\ \mathit{cons}\ x\ \mathit{xs}\ \textbf{else}\ \mathit{xs})\ \mathit{nil}\ \mathit{ls}) \\
& \quad\ \ (\lambda x\ \mathit{xs} \to \mathit{cons'}\ (x \uparrow 2)\ \mathit{xs}) \\
& \quad\ \ \mathit{nil'}) \\
\equiv\ & \quad \{\ \beta\text{-reductie}\ \} \\
& \lambda \mathit{ls} \to \mathit{sum}
\end{aligned}
$$

$$
\begin{aligned}
&\quad (\mathit{build}\ \$\ \lambda \mathit{cons}'\ \mathit{nil}' \rightarrow \\
&\qquad \mathit{foldr}\ (\lambda x\ \mathit{xs} \rightarrow \textbf{if}\ \mathit{odd}\ x\ \textbf{then}\ \mathit{cons}'\ (x \uparrow 2)\ \mathit{xs}\ \textbf{else}\ \mathit{xs})\ \mathit{nil}'\ \mathit{ls}) \\
\equiv &\quad \{\ \text{inline } \mathit{sum}\ \} \\
&\ \lambda \mathit{ls} \rightarrow \mathit{foldr}\ (+)\ 0 \\
&\quad (\mathit{build}\ \$\ \lambda \mathit{cons}'\ \mathit{nil}' \rightarrow \\
&\qquad \mathit{foldr}\ (\lambda x\ \mathit{xs} \rightarrow \textbf{if}\ \mathit{odd}\ x\ \textbf{then}\ \mathit{cons}'\ (x \uparrow 2)\ \mathit{xs}\ \textbf{else}\ \mathit{xs})\ \mathit{nil}'\ \mathit{ls}) \\
\equiv &\quad \{\ \text{foldr/build-fusion}\ \} \\
&\ \lambda \mathit{ls} \rightarrow (\lambda \mathit{cons}'\ \mathit{nil}' \rightarrow \\
&\quad \mathit{foldr}\ (\lambda x\ \mathit{xs} \rightarrow \textbf{if}\ \mathit{odd}\ x\ \textbf{then}\ \mathit{cons}'\ (x \uparrow 2)\ \mathit{xs}\ \textbf{else}\ \mathit{xs})\ \mathit{nil}'\ \mathit{ls})\ (+)\ 0 \\
\equiv &\quad \{\ \beta\text{-reductie}\ \} \\
&\ \lambda \mathit{ls} \rightarrow \mathit{foldr}\ (\lambda x\ \mathit{xs} \rightarrow \textbf{if}\ \mathit{odd}\ x\ \textbf{then}\ (x \uparrow 2) + \mathit{xs}\ \textbf{else}\ \mathit{xs})\ 0\ \mathit{ls}
\end{aligned}
$$

Uiteindelijk is *sumOfSquaredOdds*′ dus volledig gereduceerd tot één enkele *foldr* over een lijst: het is niet meer nodig om tijdelijke lijsten te alloceren om het resultaat te berekenen. In sectie 6.4 tonen we aan dat dit leidt tot significante versnellingen.

We krijgen dus als het ware het beste van beide werelden: we kunnen elegante definities gebruiken voor de functies, die eenvoudiger leesbaar zijn en makkelijker onderhoudbaar; maar tevens worden deze vertaald door de compiler tot snelle, geoptimaliseerde versies.

2.4.3 Foldr/build-fusion voor algebraïsche datatypes

In Sectie 2.3 toonden we aan dat we een fold kunnen definiëren voor om het even welk algebraïsch datatype. Dit is ook mogelijk voor *build*. Beschouw bijvoorbeeld een build voor ons *Tree*-datatype:

$$
\begin{aligned}
&\mathit{buildTree} :: (\forall b.(a \rightarrow b) \rightarrow (b \rightarrow b \rightarrow b) \rightarrow b) \rightarrow \mathit{Tree}\ a \\
&\mathit{buildTree}\ g = g\ \mathit{Leaf}\ \mathit{Branch}
\end{aligned}
$$

Zodra we beschikken over een fold en een build voor een algebraïsch datatype, is het mogelijk om fusion toe te passen. Om dit duidelijker te maken, illustreren we dit nu voor het type *Tree*. We krijgen voor *Tree* de fusion-regel gegeven in definitie 2.4.1.

Definitie 2.4.1.

$$
\mathit{foldTree}\ \mathit{leaf}\ \mathit{branch}\ (\mathit{buildTree}\ g) \quad \equiv \quad g\ \mathit{leaf}\ \mathit{branch}
$$

Het bewijs hiervan verloopt analoog aan het bewijs voor 2.4.1 en wordt hier achterwege gelaten.

Om dit te verduidelijken kunnen we kijken naar een concreet voorbeeld. Beschouw de voorbeeldfunctie *treeUpTo* die een boom maakt met alle elementen van n tot en met m in-order in de bladeren.

```
treeUpTo :: Int → Int → Tree Int
treeUpTo n m = buildTree $ λleaf branch →
  let g lo hi
      | lo ⩾ hi      = leaf lo
      | otherwise =
        let mid = (lo + hi) `div` 2
        in branch (g lo mid) (g (mid + 1) hi)
  in g n m
```

Nu kunnen we bestuderen wat er door fusion gebeurt met een expressie zoals bijvoorbeeld *sumTree* (*treeUpTo n m*), die een tijdelijke boom aanmaakt.

```
  sumTree (treeUpTo n m)
≡   { inline sumTree }
  foldTree id (+) (treeUpTo n m)
≡   { inline makeTree }
  foldTree id (+) (buildTree $ λleaf branch →
    let g lo hi
        | lo ⩾ hi      = leaf lo
        | otherwise =
          let mid = (lo + hi) `div` 2
          in branch (g lo mid) (g (mid + 1) hi)
    in g n m)
≡   { foldTree/buildTree-fusion }
  (λleaf branch →
    let g lo hi
        | lo ⩾ hi      = leaf lo
        | otherwise =
          let mid = (lo + hi) `div` 2
          in branch (g lo mid) (g (mid + 1) hi)
    in g n m)
  id (+)
≡   { β-reductie }
  let g lo hi
      | lo ⩾ hi      = id lo
      | otherwise =
        let mid = (lo + hi) `div` 2
        in (g lo mid) + (g (mid + 1) hi)
  in g n m
```

We krijgen een expressie die rechtstreeks de som uitrekent zonder ooit een constructor te gebruiken. Opnieuw zal dit voor een significante versnelling zorgen (zie sectie 6.4).

Omdat naast fold- ook build-functies eenvoudig af te leiden zijn vanuit de definitie van een datatype, hebben we dit ook geautomatiseerd. De programmeur dient enkel nog *deriveBuild* op te roepen:

```
$ (deriveBuild `Tree "buildTree")
```

Het algoritme om een build te genereren werkt als volgt:

1. De fold gebruikt een universeel gekwantificeerd type b in een functie g en geeft een waarde terug van het opgegeven type.

   $buildTree :: (\forall b.... \to b) \to Tree\ a$
   $buildTree\ g = \ldots$
   $buildList :: (\forall b.... \to b) \to [a]$
   $buildList\ g = \ldots$

2. Opnieuw krijgen we voor elke constructor een functieparameter, ditmaal voor g. De types voor deze functieparameters worden afgeleid op dezelfde manier als in het algoritme voor *deriveFold* (zie sectie 2.3).

   $buildTree :: (\forall b.(a \to b) \to (b \to b \to b) \to b) \to Tree\ a$
   $buildTree\ g = \ldots$
   $buildList :: (\forall b.(a \to b \to b) \to b \to b) \to [a]$
   $buildList\ g = \ldots$

3. De implementatie bestaat vervolgens uit het toepassen van g op de concrete constructoren.

   $buildTree :: (\forall b.(a \to b) \to (b \to b \to b) \to b) \to Tree\ a$
   $buildTree\ g = g\ Leaf\ Branch$
   $buildList :: (\forall b.(a \to b \to b) \to b \to b) \to [a]$
   $buildList\ g = g\ (:)\ [\,]$

2.4.4 Foldr/foldr-fusion

Een alternatieve vorm van fusie die eveneens van toepassing is op onze thesis is *foldr/foldr-fusion*. We kunnen dit best uitleggen door middel van een voorbeeld. Beschouw de volgende functie:

$$
\begin{aligned}
&\mathit{mean} :: [\mathit{Int}] \to \mathit{Double} \\
&\mathit{mean}\ \mathit{xs} = \mathit{fromIntegral}\ (\mathit{sum}\ \mathit{xs})\ /\ \mathit{fromIntegral}\ (\mathit{length}\ \mathit{xs})
\end{aligned}
$$

Deze eenvoudige functie berekent het gemiddelde van een lijst. Ze is gedefinieerd op elegante wijze maar is niet zeer efficiënt: de lijst *xs* wordt immers tweemaal geconsumeerd.

In een lazy taal als Haskell kan deze inefficiëntie naast onnodige tijdscomplexiteit ook extra geheugencomplexiteit met zich meebrengen. Omdat de lijst tweemaal geconsumeerd wordt, kan deze immers niet worden vrijgegeven door de garbage collector. Indien we de lijst éénmaal zouden doorlopen, zou dit uitgevoerd worden als een on-line algoritme, en is het dus niet nodig de volledige lijst in het geheugen beschikbaar te houden.

Foldr/foldr-fusion is een optimalisatie voor dergelijke gevallen. Als we dit toepassen op ons voorbeeld krijgen eerst we na inlinen van *sum* en *length* de volgende definitie:

$$
\begin{aligned}
&\mathit{mean} :: [\mathit{Int}] \to \mathit{Double} \\
&\mathit{mean}\ \mathit{xs} = \mathit{fromIntegral}\ (\mathit{foldr}\ (\lambda x\ \mathit{ys} \to x + \mathit{ys})\ 0\ \mathit{xs})\ / \\
&\quad \mathit{fromIntegral}\ (\mathit{foldr}\ (\lambda x\ \mathit{zs} \to 1 + \mathit{zs})\ 0\ \mathit{xs})
\end{aligned}
$$

In deze expressie hebben we tweemaal een *foldr* over dezelfde lijst (*xs*) – dit betekent dat we foldr/foldr-fusion kunnen toepassen. We krijgen:

$$
\begin{aligned}
&\mathit{mean}' :: [\mathit{Int}] \to \mathit{Double} \\
&\mathit{mean}'\ \mathit{xs} = \\
&\quad \textbf{let}\ (\mathit{sum}', \mathit{length}') = \mathit{foldr}\ (\lambda x\ (\mathit{ys}, \mathit{zs}) \to (x + \mathit{ys}, 1 + \mathit{zs}))\ (0, 0)\ \mathit{xs} \\
&\quad \textbf{in}\ \mathit{fromIntegral}\ \mathit{sum}'\ /\ \mathit{fromIntegral}\ \mathit{length}'
\end{aligned}
$$

In het algemeen kunnen we op deze manier twee algebra's samenvoegen tot één enkele algebra, op voorwaarde dat ze op hetzelfde lijst-type werken, in ons voorbeeld $[a]$:

$$
\begin{aligned}
&c_1 :: a \to B_1 \to B_1 \\
&c_2 :: a \to B_2 \to B_2 \\
&n_1 :: B_1 \\
&n_2 :: B_2
\end{aligned}
\quad \Leftrightarrow \quad
\begin{aligned}
&c_{12} :: a \to (B_1, B_2) \to (B_1, B_2) \\
&c_{12} = \lambda x\ (\mathit{ys}, \mathit{zs}) \to (c_1\ x\ \mathit{ys}, c_2\ x\ \mathit{zs}) \\
&n_{12} :: (B_1, B_2) \\
&n_{12} = (n_1, n_2)
\end{aligned}
$$

We kunnen deze optimalisatie verschillende keren na elkaar uitvoeren voor de gevallen waar we meer dan twee keer dezelfde lijst consumeren met een

foldr: in die gevallen krijgen we types met geneste tuples, zoals bijvoorbeeld $((B_1, B_2), B_3)$.

Eveneens is deze optimalisatie uitbreidbaar tot andere recursieve algebraïsche datatypes naast lijsten. Deze extensie volgt natuurlijk eens de fold voor een dergelijk datatype gedefinieerd is. We beperken ons hier tot een klein voorbeeld: het berekenen van de gemiddelde waarde uit een boom.

$$
\begin{aligned}
&\mathit{meanTree} :: \mathit{Tree\ Int} \to \mathit{Double} \\
&\mathit{meanTree\ tree} = \\
&\quad \textbf{let}\ (\mathit{sum}', \mathit{size}) = \mathit{foldTree}\ (\lambda x \to (x, 1)) \\
&\qquad\qquad\qquad\qquad\qquad\quad (\lambda(xl, xr)\ (yl, yr) \to (xl + yl, xr + yr)) \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{tree} \\
&\quad \textbf{in}\ \mathit{fromIntegral\ sum}' \ / \ \mathit{fromIntegral\ size}
\end{aligned}
$$

In tegenstelling tot foldr/build-fusion zorgt deze optimalisatie echter vaak niet voor een snellere uitvoering van het programma. Dit komt omdat er een overhead is geassocieerd met het alloceren van tuples – en voor kleine lijsten kan de vertraging door deze overhead de snelheidswinst van de optimalisatie neutraliseren of zelfs zorgen voor een algemene vertraging.

Omwille van deze reden besloten we in onze proof-of-concept implementatie te kiezen voor foldr/build-fusion. We moeten echter wel opmerken dat integratie van ons werk in een systeem dat reeds foldr/foldr-fusion gebruikt zou leiden tot een verbetering van de efficiëntie: aangezien wij expliciet recursieve functies (die momenteel niet kunnen genieten van foldr/foldr-fusion) herschrijven naar functies die gebruik maken van fold, zullen er meer opportuniteiten zijn om deze optimalisatie toe te passen.

3 Detectie van folds

3.1 Notatie

Om de uitleg en regels in dit hoofdstuk en hoofdstuk 4 eenvoudiger te maken, maken we geen gebruik van de normale Haskell-syntax, noch GHC Core (zie sectie 5.1). In plaats daarvan gebruiken we de simpele, ongetypeerde lambda-calculus, uitgebreid met constructoren, pattern matching, en recursieve bindings.

Deze syntax wordt gegeven door:

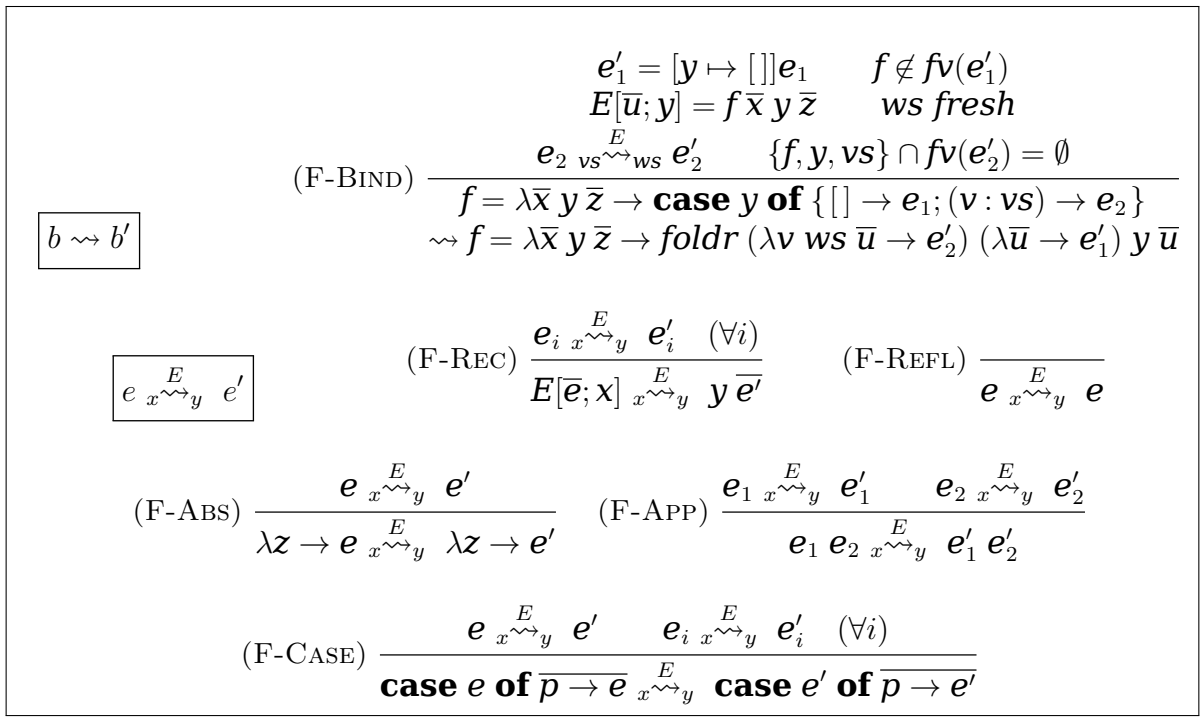
$$
\begin{array}{lrcl}
\text{binding} & b & ::= & x = e \\
\text{pattern} & p & ::= & K\,\overline{x} \\
\text{expression} & e & ::= & x \\
 & & | & e\ e \\
 & & | & \lambda x \to e \\
 & & | & K \\
 & & | & \textbf{case}\ e\ \textbf{of}\ \overline{p \to e}
\end{array}
$$

Het is eenvoudig te zien hoe we deze in de praktijk kunnen uitbreiden tot de volledige GHC Core syntax.

Eveneens hebben we een *context* nodig:

$$
\begin{array}{rcl}
E & ::= & x \\
 & | & E\,x \\
 & | & E\,\square \\
 & | & E\,\triangle
\end{array}
$$

Een dergelijke context E stelt een functie voor die toegepast wordt op een aantal argumenten. De functie en een aantal argumenten zijn reeds bekend.



Figuur 3.1: De niet-deterministische regels die we gebruiken om mogelijke folds te ontdekken en te herschrijven.

Voor de andere argumenten zijn er *gaten* die nog kunnen worden ingevuld door expressies. We onderscheiden twee soorten gaten, aangegeven met de symbolen $\Box$ en $\triangle$. Een $\Box$ geeft een onbelangrijk argument aan en een $\triangle$ duidt op een argument dat in het bijzonder de aandacht verdient (concreet zal dit in ons geval de waarde zijn waarover we folden).

De functie $E[\overline{e}; e]$ past deze context E toe door de gaten op te vullen met de gegeven expressies.

$$\begin{array}{lcl} x[\epsilon; e] & = & x \\ (E\ x)[\overline{e}; e] & = & E[\overline{e}; e]\,x \\ (E\ \Box)[\overline{e}, e_1; e] & = & E[\overline{e}; e]\,e_1 \\ (E\ \triangle)[\overline{e}; e] & = & E[\overline{e}; e]\,e \end{array}$$

Merk hierbij op dat we er vanuit gaan dat het aantal $\Box$ gaten precies gelijk is aan het aantal meegegeven expressies $\overline{e}$.

3.2 Regels voor de detectie van folds

De regels die we gebruiken zijn te zien in Figuur 3.1. Deze figuur is specifiek voor folds over lijsten, m.a.w. *foldr*. Op die manier kunnen we de uitleg zo simpel mogelijk te houden. In sectie 3.4 zien we hoe dit kan worden uitgebreid tot andere algebraïsche datatypes.

Functies met één enkel argument We hebben een relatie $b \rightsquigarrow b'$ (van het type $Bind \times Bind$). Deze relatie legt een verband tussen expliciet recursieve functies en de corresponderende functies herschreven in termen van *foldr*. De relatie $b \rightsquigarrow b'$ maakt gebruik van de enkele regel F-BIND. Om deze regel te verduidelijken kijken we eerst naar een gespecialiseerde regel, F-BIND'. Deze gespecialiseerde regel is enkel van toepassing op functies met één enkel argument.

$$\text{(F-BIND')}\ \frac{\begin{array}{c} e_1' = [y \mapsto [\,]]e_1 \qquad f \notin fv(e_1) \qquad ws\ fresh \\ e_2\ {}_{vs}\overset{f\ \triangle}{\rightsquigarrow}_{ws}\ e_2' \qquad \{f, y, vs\} \cap fv(e_2') = \emptyset \end{array}}{\begin{array}{c} f = \lambda y \to \textbf{case}\ y\ \textbf{of}\ \{[\,] \to e_1; (v : vs) \to e_2\} \\ \rightsquigarrow f = \lambda y \to foldr\ (\lambda v\ ws \to e_2')\ e_1'\ y \end{array}}$$

Deze regel zet een functie zoals:

$$\begin{array}{l} sum = \lambda y \to \textbf{case}\ y\ \textbf{of} \\ \quad [\,] \qquad \to 0 \\ \quad (v : vs) \to (+)\ v\ (sum\ vs) \end{array}$$

om naar:

$$sum = \lambda y \to foldr\ (\lambda v\ ws \to (+)\ v\ ws)\ 0\ y$$

Deze omzetting verloopt door op een zeer eenvoudige manier de regels toe te passen. Het belangrijkste hierbij is de recursieve oproepen in e_2 te vervangen door de variabele *ws*. Het verband tussen de oorspronkelijke en herschreven expressie wordt bepaald door de relatie:

$$e\ {}_{x}\overset{E}{\rightsquigarrow}_{y}\ e'$$

Deze relatie maakt gebruik van vijf verschillende regels. F-REC is verantwoordelijk voor het effectieve herschrijven van recursieve oproepen. Voor

andere expressies gebruiken we ofwel één van de drie herschrijfregels F-ABS, F-APP, F-CASE, ofwel de reflectieve regel F-REFL, die de expressie gewoon behoudt. In het vereenvoudigde geval, waarbij we slechts één argument hebben, kan F-REC gereduceerd worden tot:

$$\text{(F-REC')}\ \frac{}{f\ x\ \overset{f\ \triangle}{\underset{x}{\rightsquigarrow}}_{y}\ y}$$

Bijgevolg krijgen we: $sum\ vs\ \overset{sum\ \triangle}{\underset{vs}{\rightsquigarrow}}_{ws}\ ws$.

We hebben ook een reeks belangrijke voorwaarden bij deze regel. Deze dienen ertoe te verzekeren dat de functie wel degelijk een catamorfisme is, zodanig dat de gegenereerde fold een geldige expressie is. We lichten het specifieke doel van de voorwaarden kort toe:

- Indien *vs* nog voorkomt in e'_2, kan dit niet onder de vorm *f vs* zijn – dan zou dit namelijk vervangen geweest zijn door onze regels. Indien het in andere vorm voorkomt, dan is de functie geen catamorfisme, maar een *paramorfisme*. Een voorbeeld van een dergelijk paramorfisme is de functie:

  $$\begin{aligned} &suffixes = \lambda y \to \textbf{case}\ y\ \textbf{of} \\ &\quad [\,] \qquad \to [\,] \\ &\quad (v : vs) \to vs : suffixes\ vs \end{aligned}$$

  Paramorfismen kunnen worden geschreven met behulp van de hogere-orde functie *para*:

  $$\begin{aligned} &para :: (a \to [a] \to b \to b) \to b \to [a] \to b \\ &para\ f\ z\ [\,] \qquad = z \\ &para\ f\ z\ (x : xs) = f\ x\ xs\ (para\ f\ z\ xs) \end{aligned}$$

  In het geval van *suffixes* krijgen we:

  $$suffixes' = para\ (\lambda v\ vw\ ws \to vw : ws)\ [\,]$$

  Om dezelfde reden mag *y* niet voorkomen in e'_2. In het geval van de (:) constructor, is *y* equivalent aan $(v : vs)$. Indien we dus nog een voorkomen van *y* hebben, impliceert dit een voorkomen van *vs* – en we vermelden hierboven al waarom we dit niet kunnen toelaten voor catamorfismes.

  In het geval van de $[\,]$ constructor, vervangen we *y* door $[\,]$ via de regel F-BIND', en vormt dit dus geen probleem.

- Als f voorkomt in een andere vorm dan recursieve calls van de vorm $f\,vs$, dan is de functie mogelijks geen catamorfisme. Beschouw bijvoorbeeld de functie, die zal resulteren in oneindige recursie wanneer er een argument anders dan de lege lijst aan wordt meegegeven:

$$\begin{aligned} f = \lambda x \to \textbf{case}\ x\ \textbf{of} \\ \quad [\,] \quad\ \to 0 \\ \quad (v : vs) \to v + f\,vs + f\,[1,2,3] \end{aligned}$$

Functies met meerdere argumenten Nu de F-BIND' regel duidelijk is, kunnen we de meer uitgebreide regel F-BIND bespreken. Deze regel laat bijkomende argumenten toe, naast de waarde waarover we folden. Deze bijkomende argumenten kunnen zowel voorkomen voor als na het scrutinee-argument y.

We onderscheiden hier twee klasses argumenten: statische en veranderlijke argumenten. Laat ons beginnen bij de eenvoudigste klasse, statische argumenten.

Een voorbeeld hiervan is het argument f in de functie map:

$$\begin{aligned} map = \lambda f\,y \to \textbf{case}\ y\ \textbf{of} \\ \quad [\,] \quad\ \to [\,] \\ \quad (v : vs) \to (:)\ (f\,v)\ (map\,f\,vs) \end{aligned}$$

Deze parameter blijft dezelfde in elke recursieve oproep van map – vandaar de naam statische argumenten. De regel F-BIND vermeldt deze veranderlijke argumenten niet rechtstreeks, maar ze worden bijgehouden in de context E (zie sectie 3.1).

Voor de map-functie hierboven is deze context bijvoorbeeld $map\,f\,\triangle$.

De tweede klasse, veranderlijke argumenten, dienen we op een andere manier aan te pakken. Een typisch voorbeeld van veranderlijke argumenten zijn catamorfismes die een accumulator-parameter gebruiken. Beschouw bijvoorbeeld de functie:

$$\begin{aligned} suml = \lambda y\,acc \to \textbf{case}\ y\ \textbf{of} \\ \quad [\,] \quad\ \to acc \\ \quad (v : vs) \to suml\ vs\ (v + acc) \end{aligned}$$

De regel F-BIND verwijst naar deze argumenten als $\overline{u}$ en maakt hiervoor $\square$ gaten in de context E. Voor de functie $suml$ is deze context bijvoorbeeld $suml\ \triangle\ \square$.

Het feit dat deze argumenten kunnen veranderen in elke recursiestap, betekent dat we deze telkens opnieuw moeten doorgeven. Dit doen we door ze in de anonieme functies door te geven als extra argumenten, in de regel aangegeven als $\lambda\overline{u} \to \ldots$. De initiële waarden hiervoor (de argumenten doorgegeven aan de oorspronkelijke functie) moeten vervolgens ook worden meegegeven aan het resultaat van *foldr*.

Het is belangrijk dat we bij de veranderlijke argumenten in elke stap van de recursieve oproep de oude waarden vervangen door de nieuwe waarden. Hiertoe dient de regel F-Rec. De nieuwe waarden van de veranderlijke argumenten worden aangegeven door $\overline{e}$. Met behulp van de context E kunnen we deze dan invullen in de de anonieme functie, waar geen expliciete recursie voorkomt. De recursieve oproep, van de vorm $E[\overline{e}; vs]$, wordt herschreven naar $ws\ \overline{e}$. Op die manier worden de veranderlijke argumenten meegegeven aan het resultaat van de (impliciete) recursieve oproep, ws.

Als we opnieuw *suml* als voorbeeld nemen, krijgen we nu:

$$\begin{aligned} suml = \lambda y\ acc \to foldr\ &(\lambda v\ ws\ acc \to ws\ (v + acc)) \\ &(\lambda acc \to acc)\ y\ acc \end{aligned}$$

Een klein maar belangrijk detail is dat de regel F-Rec de veranderlijke argumenten $\overline{e}$ ook zal herschrijven, door de regels toe te passen op een recursieve manier. De veranderlijke argumenten kunnen immers ook expliciet recursieve oproepen bevatten die we dienen om te zetten. Beschouw bijvoorbeeld de functie:

$$\begin{aligned} f = \lambda y\ acc \to\ &\mathbf{case}\ y\ \mathbf{of} \\ &[\,] \quad\ \to acc \\ &(v : vs) \to f\ vs\ (f\ vs\ (v + acc)) \end{aligned}$$

Die aldus herschreven wordt als:

$$\begin{aligned} f = \lambda y\ acc \to\ &foldr\ (\lambda v\ ws\ acc \to ws\ (ws\ (v + acc))) \\ &(\lambda acc \to acc)\ y\ acc \end{aligned}$$

Waarbij we het geneste recursieve voorkomen van ws opmerken, wat overeenkomt met de geneste oproep van f in de oorspronkelijke functie.

3.3 Gedegenereerde folds

De herschrijfregels die we bespraken in sectie 3.2 herschrijven ook bepaalde niet-recursieve functies als folds. Beschouw bijvoorbeeld de bekende

functie *head*:

$$
\begin{aligned}
&\mathit{head} :: [a] \to a \\
&\mathit{head} = \lambda l \to \textbf{case}\ l\ \textbf{of} \\
&\quad [\,] \qquad \to \mathit{error}\ \texttt{"empty list"} \\
&\quad (x : xs) \to x
\end{aligned}
$$

Door toepassing van de herschrijfregels krijgen we:

$$
\begin{aligned}
&\mathit{head} :: [a] \to a \\
&\mathit{head} = \lambda l \to \mathit{foldr}\ (\lambda x\ xs \to x)\ (\mathit{error}\ \texttt{"empty list"})\ l
\end{aligned}
$$

Deze *gedegenereerde* folds zijn niet relevant voor deze thesis. Het is immers niet interessant om deze functies te beschouwen voor foldr/build-fusion: andere eenvoudige technieken zoals inlining en *case specialization* volstaan.

Een bijkomend argument is dat de herschreven versie, in termen van *foldr*, ook moeilijker is om te begrijpen is dan de oorspronkelijke versie. Programmeurs bekend met de *foldr* functie een verwachten hier namelijk een vorm van recursie (die er hier niet is).

Gelukkig kunnen we eenvoudig bepalen of een functie al dan niet een gedegenereerde fold is. Als we regel F-Rec minstens éénmaal gebruikten, is er zeker sprake van recursie. Anders is de functie in kwestie een gedegenereerde en verkiezen we om de oorspronkelijke definitie te gebruiken in plaats van de herschreven definitie, die gebruikt maakt van *foldr*.

3.4 Detectie van folds over andere algebraïsche datatypes

Een volledig gesloten verzameling van regels hoe we expliciete recursie over een willekeurig, gegeven datatype kunnen herschrijven naar een fold over dat datatype zou ons te ver leiden. Daarom geven we in deze sectie een minder formele uitleg over hoe we de regels hiertoe kunnen uitbreiden. Let er wel op dat onze implementatie deze omzetting ook implementeerd (zie subsectie 5.3.1).

Ter illustratie gebruiken we hier de eenvoudige expliciet recursieve functie *sumTree*:

$$
\begin{aligned}
&\mathit{sumTree} :: \mathit{Tree\ Int} \to \mathit{Int} \\
&\mathit{sumTree}\ (\mathit{Leaf}\ x) \qquad = x \\
&\mathit{sumTree}\ (\mathit{Branch}\ l\ r) = \mathit{sumTree}\ l + \mathit{sumTree}\ r
\end{aligned}
$$

De regel F-BIND is specifiek voor lijsten, verwijst onder meer letterlijk naar de constructoren $(:)$ en $[]$. Om te illustreren hoe we deze regel kunnen uitbreiden, vestigen we de aandacht op drie belangrijke veranderingen:

- We krijgen n constructoren in plaats van de twee constructoren van een lijst. We hebben dus ook n **case**-alternatieven: e_1, e_2, $\ldots e_n$.
- De functie *foldr* wordt uiteraard vervangen door de fold van het gegeven datatype. Voor *sumTree* hebben we het type *Tree*, dus krijgen we de fold *foldTree*.
- Bij een lijst is *vs* de enige recursieve subterm die kan optreden. In andere recursieve algebraïsche datatypes kunnen dit er meerdere zijn. Zo hebben we bij het type *Tree* de recursieve subtermen l en r, beide in de *Branch* constructor.

Dit laatste impliceert ook dat de relatie tussen expressies van een andere vorm zal zijn. We krijgen voor het type *Tree*:

$$e \ {}_{l,r}\overset{E}{\rightsquigarrow}_{l',r'} \ e'$$

Met E opnieuw een context die recursieve oproepen naar sumtree herkent. Concreet is dit voor *sumTree*:

$$E = sumTree \ \triangle$$

Behalve deze verandering in de vorm van de relatie, hoeven we de regels F-REFL, F-ABS, F-APP en F-CASE niet te veranderen. Deze staan immers al in een algemene vorm en verwijzen niet concreet naar het lijst-datatype. We moeten de regel F-REC wel uitbreiden voor het type *Tree*: er zijn nu immers twee recursieve oproepen mogelijk. We krijgen de regel F-REC-TREE:

$$\text{(F-REC-TREE)} \ \frac{e_i \ {}_{l,r}\overset{E}{\rightsquigarrow}_{l',r'} \ e_i' \quad (\forall i)}{E[\overline{e}; l] \ {}_{l,r}\overset{E}{\rightsquigarrow}_{l',r'} \ l' \ \overline{e'} \qquad E[\overline{e}; r] \ {}_{l,r}\overset{E}{\rightsquigarrow}_{l',r'} \ r' \ \overline{e'}}$$

Dit laat ons toe om zowel de recursieve oproep *sumTree l* als *sumTree r* te herschrijven als respectievelijk l' en r'. Voor *sumTree* krijgen na toepassing van deze regels uiteindelijk de herschreven versie:

$$sumTree :: Tree \ Int \to Int$$
$$sumTree \ y = foldTree \ (\lambda x \to x) \ (\lambda l' \ r' \to l' + r') \ y$$

4 Detectie van builds

In Figuur 4.1 geven we de niet-deterministische regels die we gebruiken om builds te herkennen. Deze regels zijn opnieuw specifiek voor lijsten, teneinde de uitleg te vereenvoudigen.

De relatie $b \rightarrowtail b'; b_g$ staat centraal. Deze legt het verband tussen de binding b en de bindings $b'; b_g$. De binding b maakt expliciet gebruik van de concrete constructoren, en b' is een herschreven variant die gebruik maakt van de functie *build*. Bijkomend krijgen we b_g, een binding die gebruikt wordt als de generatorfuctie (meestal g genoemd). Deze wordt gegeven als argument van *build*.

Deze relatie maakt gebruik van één enkele regel, namelijk B-BIND. Deze regel herschrijft de de definitie van een functie en definiëert ook de bijkomende functie g. De functie die we herschrijven mag om het even hoeveel argumenten hebben – deze worden voorgesteld door $\lambda\overline{x} \rightarrow \ldots$.

Deze argumenten worden ook meegegeven aan de generatorfuctie g. Op deze manier kan g op dezelfde manier als de oorspronkelijke functie een waarde opbouwen – behalve dat er nu abstracte versies van de constructoren gebruikt worden. Een voorbeeld van een dergelijke functie met argumenten is *map*. Als we *map* herschrijven via onze regels krijgen we:

$$
\begin{aligned}
map &= \lambda f \rightarrow \lambda l \rightarrow build\ (g\ f\ l) \\
g \quad &= \lambda f \rightarrow \lambda l \rightarrow \lambda c \rightarrow \lambda n \rightarrow \\
&\quad\ \textbf{case}\ l\ \textbf{of} \\
&\qquad [\,] \qquad\ \rightarrow n \\
&\qquad (y : ys) \rightarrow c\ (f\ y)\ (g\ f\ ys\ c\ n)
\end{aligned}
$$

Hierbij hebben we $\overline{x} = [f, l]$. We ook zien dat f een statisch argument is en l een veranderlijk argument. In tegenstelling tot de herkenning van folds (zie sectie 3.2) moeten we nu geen onderscheid maken tussen beide.

Om de expressies in de bindings te herschrijven maken we gebruik van de

$$\boxed{b \rightarrowtail b'; b_g} \qquad \text{(B-Bind)}\ \frac{\begin{array}{c} c, n, g \text{ fresh} \\ e \ {}_f\overset{c,n}{\rightarrowtail}_g \ e' \end{array}}{\begin{array}{l} f = \lambda \overline{x} \to e \ \rightarrowtail \\ f = \lambda \overline{x} \to \mathit{build}\ (g\ \overline{x}); \\ g = \lambda \overline{x} \to \lambda c \to \lambda n \to e' \end{array}}$$

$$\boxed{e \ {}_f\overset{c,n}{\rightarrowtail}_g \ e'} \qquad \text{(B-Rec)}\ \frac{}{f\ \overline{e}\ {}_f\overset{c,n}{\rightarrowtail}_g \ g\ \overline{e}\ c\ n}$$

$$\text{(B-Nil)}\ \frac{}{[\,]\ {}_f\overset{c,n}{\rightarrowtail}_g \ n} \qquad \text{(B-Cons)}\ \frac{e_2\ {}_f\overset{c,n}{\rightarrowtail}_g \ e_2'}{(e_1 : e_2)\ {}_f\overset{c,n}{\rightarrowtail}_g \ c\ e_1\ e_2'}$$

$$\text{(B-Build)}\ \frac{}{\mathit{build}\ e\ {}_f\overset{c,n}{\rightarrowtail}_g \ e\ c\ n}$$

$$\text{(B-Case)}\ \frac{e_i\ {}_f\overset{c,n}{\rightarrowtail}_g \ e_i' \quad (\forall i)}{\textbf{case}\ e\ \textbf{of}\ \overline{p \to e}\ {}_f\overset{c,n}{\rightarrowtail}_g \ \textbf{case}\ e\ \textbf{of}\ \overline{p \to e'}}$$

Figuur 4.1: Onze regels voor het herkennen van builds

volgende relatie:

$$e \ {}_f\overset{c,n}{\rightarrowtail}_g \ e'$$

Deze relatie resulteert in de definitie van de generatorfuctie g. We maken hierbij gebruik van vijf verschillende regels. Hiervan behandelen de eerste vier regels al de manieren waarop we het aanmaken van een lijst kunnen herkennen. De vijfde regel, tenslotte, breidt de herkenning uit zodanig dat we ook **case**-expressies kunnen herschrijven, op voorwaarde dat we alle **case**-alternatieven kunnen herschrijven.

1. De meest eenvoudige manier om een lijst aan te maken is simpelweg de constructor voor een lege lijst, $[\,]$. Via de regel B-Nil herschrijven we deze constructor naar n, de abstracte versie van $[\,]$ die wordt meegegeven als argument aan g.

2. Eveneens moeten we de $(:)$ constructor herschrijven. Hiertoe dient de regel B-Cons. We vervangen $(:)$ door de functie c, die wordt meegegeven aan g als argument. Dit is echter niet voldoende: het tweede argument van $(:)$ is de tail van de lijst, en deze lijst moet ook opgebouwd

worden gebruik makende van n en c in plaats van $[]$ en $(:)$. Daarom herschrijven we ook de tail van de lijst, door de regels op een recursieve manier toe te passen.

3. Indien er recursieve oproepen voorkomen naar de oorspronkelijke functie, moeten deze herschreven worden naar recursieve oproepen naar de nieuwe generatorfuctie g. Hiertoe dient de regel B-REC.

4. De regel B-BUILD handelt het geval af waarin we een geneste oproep naar *build* vinden. Deze *build* is dan van de vorm *build* g' – met g' een andere generatorfunctie. Deze g' is van de vorm:

   $$g' = \lambda c'\ n' \to \ldots$$

   We willen nu deze *build* g' herschrijven zodanig dat deze ook onze argumenten c en n gebruikt. Dit gaat op zeer eenvoudige manier: namelijk, $g'\ c\ n$.

In het *map*-voorbeeld hierboven illustreerden we reeds alle regels behalve B-BUILD. Hiervan geven we nu een voorbeeld. Beschouw de volgende functies:

```
toFront :: Eq a ⇒ a → [a] → [a]
toFront y ys = y : filter (≢ y) ys
```

Veronderstel dat *filter* reeds herschreven is in termen van *build*, m.a.w., we hebben:

```
filter :: (a → Bool) → [a] → [a]
filter f ls = build $ λcons nil →
  foldr (λx xs → if f x then cons x xs else xs) nil ls
```

Als we nu *filter* inlinen in de definitie van *toFront* krijgen we:

```
toFront :: Eq a ⇒ a → [a] → [a]
toFront y ys = y : (build $ λcons nil →
  foldr (λx xs → if (≢ y) x then cons x xs else xs) nil ys)
```

Deze expressie kan nu worden herschreven door gebruik te maken van de regel B-BUILD:

```
toFront :: Eq a ⇒ a → [a] → [a]
toFront y ys = build (g y ys)
g = λy ys c n → c y ((λcons nil →
  foldr (λx xs → if (≢ y) x then cons x xs else xs) nil ys) c n)
```

4.1 Gedegenereerde builds

Net zoals we niet-recursieve catamorfismes de naam gedegenereerde folds gaven, noemen we niet-recursieve builds gedegenereerde builds. Beschouw als voorbeeld:

$$
\begin{aligned}
&f :: [Int] \\
&f = 1 : 2 : 3 : [\,]
\end{aligned}
$$

Via de regels B-Nil en B-Cons wordt deze functie herschreven tot:

$$
\begin{aligned}
&f :: [Int] \\
&f = build\ g \\
&g = \lambda c\ n \to c\ 1\ (c\ 2\ (c\ 3\ n))
\end{aligned}
$$

Een dergelijke build kunnen we eenvoudig herkennen door de afwezigheid van een recursieve oproep. In ons algoritme gebeurt dit dus door bij te houden of de regel B-Rec al dan niet gebruikt wordt tijdens het herschrijven van de functie.

Strikt gezien hebben we geen foldr/build-fusion nodig om dit te optimaliseren: hiertoe volstaat specialisatie[1] en inlinen van de consumerende functie, bijvoorbeeld *sum*.

$$
\begin{aligned}
& sum\ f \\
\equiv\ & \quad \{\ \text{inline } f\ \} \\
& sum\ (1 : 2 : 3 : [\,]) \\
\equiv\ & \quad \{\ \text{inline } sum\ (:)\ \} \\
& 1 + sum\ (2 : 3 : [\,]) \\
\equiv\ & \quad \{\ \text{inline } sum\ (:) \text{ nog 3 maal }\} \\
& 1 + 2 + 3 + sum\ [\,] \\
\equiv\ & \quad \{\ \text{inline } sum\ [\,]\ \} \\
& 1 + 2 + 3 + 0
\end{aligned}
$$

In de praktijk gaat GHC echter niet op een dergelijke manier agressief inlinen, zelfs niet wanneer de `-O2` vlag wordt meegegeven. Daarom is het, ondanks het ontbreken van recursie, toch nuttig om deze functies te herschrijven als build: foldr/build-fusion is dan in staat om expressies als *sum f* wel te optimaliseren.

[1] Hiermee bedoelen we *case specialization*. Er worden twee functies aangemaakt, *sum_nil* en *sum_cons*, die specifiek voor respectievelijk lege lijsten en niet-lege lijsten zijn. Vervolgens kan *sum* vervangen worden door de juiste specifieke versie als de constructor van het argument bekend is.

5 Implementatie

5.1 GHC Core

In hoofdstuk 2 beschreven we al dat we voor deze thesis werken met GHC [29], de de-facto standaard Haskell compiler.

GHC werkt met een *kerneltaal*. Een kerneltaal is een gereduceerde subset van de programmeertaal (in dit geval Haskell). Bovendien is het mogelijk om elk Haskell-programma uit te drukken in de kerneltaal.

Een dergelijke vertaling gebeurt door de compiler en is beter bekend onder de naam desugaring[1]. Programma's die uitgedrukt worden in de kerneltaal zijn meestal minder beknopt.

Het gebruik van een dergelijke kerneltaal heeft verschillende voordelen:

- De syntax van de kerneltaal is zeer beperkt. Hierdoor kunnen de programmeur en de compiler op een meer eenvoudige manier redeneren over expressies, zonder rekening te houden met op dat moment oninteressante syntactische details.
- Om nieuwe syntax toe te voegen, dient men enkel het *desugaring*-proces aan te passen en hoeft men geen aanpassingen te doen in de rest van de compiler.
- Verschillende programmeertalen kunnen dezelfde kerneltaal delen. Dit laat toe om bepaalde tools en optimalisaties éénmaal te schrijven en vervolgens toe te passen voor programma's geschreven in verschillende programmeertalen. Dit voordeel is echter niet van toepassing op GHC, omdat deze een eigen kerneltaal gebruikt.

[1] De vele syntactische structuren die in idiomatische Haskell-code gebruikt worden staan bekend als *syntactic sugar*, vandaar deze naam.

De kerneltaal van die GHC gebruikt heet GHC Core [32].

Om onze fold- en build-detectie te implementeren hebben we dus twee keuzes. We kunnen ofwel de Haskell-code direct manipuleren. Er bestaan reeds verschillende bibliotheken om deze taak eenvoudiger te maken, zoals bijvoorbeeld *haskell-src-exts* [2].

We kunnen echter ook werken met de GHC Core. Dit heeft voor ons een groot aantal voordelen.

- Zoals we eerder al vermeldden, is het syntax veel eenvoudiger. Dit drukt zich ook uit in de complexiteit van de abstracte syntaxboom: Ter illustratie: het *Expr*-type dat in *haskell-src-exts* gebruikt wordt heeft 46 verschillende constructoren, terwijl het *Expr*-type van GHC Core er slechts 10 heeft.
- De GHC Core gaat door verschillende optimalisatie-passes. Veel van deze passes vereenvoudigen de expressies, wat op zijn beurt de analyse makkelijker maakt. Beschouw bijvoorbeeld de volgende functie *jibble*:

  $$\begin{aligned} &\mathit{jibble} :: [\mathit{Int}] \to \mathit{Int} \\ &\mathit{jibble}\ [\,] \quad\ = 1 \\ &\mathit{jibble}\ (x : \mathit{xs}) = \mathit{wiggle}\ x\ \mathit{xs} \\ &\mathit{wiggle} :: \mathit{Int} \to [\mathit{Int}] \to \mathit{Int} \\ &\mathit{wiggle}\ x\ \mathit{xs} = x * \mathit{jibble}\ \mathit{xs} + 1 \end{aligned}$$

  Hier is het moeilijk om een *foldr*-patroon te herkennen door het gebruik van de hulpfunctie *wiggle*: onze implementatie gaat immers niet kijken wat de definitie van *wiggle* is. Maar, eens deze functie ge-inlined is, krijgen we de functie:

  $$\begin{aligned} &\mathit{jibble} :: [\mathit{Int}] \to \mathit{Int} \\ &\mathit{jibble}\ [\,] \quad\ = 1 \\ &\mathit{jibble}\ (x : \mathit{xs}) = x * \mathit{jibble}\ \mathit{xs} + 1 \end{aligned}$$

  Onze detector kan de laatste versie onmiddelijk herkennen.
- Tenslotte beschikken we ook over type-informatie: de GHC API laat ons toe types van onder meer variabelen en functies op te vragen. Dit is in principe niet essentieel voor onze detector, maar kan wel zeer nuttig zijn. Beschouw bijvoorbeeld:

  $$\begin{aligned} &\mathit{add} :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\ &\mathit{add}\ x\ y = \ldots \end{aligned}$$

  We kunnen, zonder de definitie van *add* te bekijken, al uit de typesignatuur opmaken dat *add* geen fold noch build zal zijn. Het is immers

niet mogelijk te folden over een *Int* of er één aan te maken met build: *Int* valt niet in de klasse van de algebraïsche datatypes.

We dienen wel op te merken dat er ook een belangrijk nadeel gekoppeld is aan het werken met GHC Core in plaats van Haskell code. Het wordt namelijk veel moeilijker om de resultaten van onze analyse te gebruiken voor *refactoring*.

In dit geval zouden we de originele code willen herschrijven onder de vorm van een fold of een build. Dit vereist echter een geannoteerde abstracte syntaxboom die toelaat om expressies uit GHC Core terug te koppelen naar Haskell expressies, inclusief alle syntactische sugar waar de programmeur gebruik van kan maken. Automatisch herschrijven van expressies in GHC Core zorgt dan voor een soortgelijke update van de corresponderende Haskell code. Deze stap valt echter buiten het huidig bereik van deze thesis.

Om bovenstaande redenen kiezen we er dus voor om met GHC Core te werken. In Figuur 5.1 geven we een kort overzicht van de omzetting van Haskell-expressies naar de corresponderende expressies in GHC Core.

5.2 Het GHC Plugins framework

Nu we beslist hebben op het niveau van GHC Core te werken, dringt zich de vraag op hoe we deze GHC Core-expressies kunnen manipuleren.

De vraag is nu hoe we deze GHC Core kunnen manipuleren. Tot voor kort was dit enkel mogelijk door de source code van GHC direct aan te passen.

Om aan dit probleem tegemoet te komen werd een nieuw pluginsysteem geïntroduceerd [31] in GHC 7.2.1, dat de praktische kant van een dergelijke manipulatie behoorlijk vereenvoudigt.

Meer bepaald is het nu mogelijk om Core-naar-Core transformaties te implementeren in aparte modules, en deze vervolgens mee te geven aan GHC via commmand-line argumenten.

De module moet een $plugin :: Plugin$ definitie bevatten.

$$plugin :: Plugin$$
$$plugin = defaultPlugin \{installCoreToDos = install\}$$

De *installCoreToDos* laat toe om de lijst van passes aan te passen. Dit is een standaard Haskell-lijst en bevat initiëel alle passes die GHC traditioneel uit-

```
"Jan"
```

```
((:) 'J' ((:) 'a' ((:) 'n')))
```

```
head []      = ⊥
head (x : _) = x
```

```
head = λxs → case xs of
  []      → ⊥
  (:) x _ → x
```

```
let x = 3
    y = 4
in x + y + z
where z = 5
```

```
let z = 5
in let x = 3
   in let y = 4
      in (+) x ((+) y z)
```

```
compare a b
  | a > b      = GT
  | a ≡ b      = EQ
  | otherwise = LT
```

```
compare = λa → λb →
  case (>) a b of
    True  → GT
    False → case (≡) a b of
      True  → EQ
      False → LT
```

```
foldM f a []      = return a
foldM f a (x : xs) = do
  a' ← f a x
  foldM f a' xs
```

```
foldM = λf a ls → case ls of
  []      → return a
  (:) x xs →
    f a x >>= λa' → foldM f a' xs
```

Figuur 5.1: Een overzicht van hoe Haskell-expressies worden omgezet naar GHC Core-expressies. Links worden de Haskell-expressies getoond, en rechts de overeenkomstige GHC Core-expressies

voert. Met *intersperse* kunnen we bijvoorbeeld onze passes laten uitvoeren tussen elke twee GHC-passes.

```
install :: [CommandLineOption] → [CoreToDo] → CoreM [CoreToDo]
install _options passes = return $ intersperse myPlugin passes
  where
    myPlugin = CoreDoPluginPass "My plugin" (bindsOnlyPass myPass)
```

De implementatie van de effectieve pass heeft typisch de type-signatuur *CoreProgram → CoreM CoreProgram*. Hierin kunnen we dus gemakkelijk de expressies bewerken, gelet op het feit dat ze worden voorgesteld als een algebraïsch datatype.

```
myPass :: CoreProgram → CoreM CoreProgram
myPass = ...
```

We illustreren hier een vereenvoudigde versie van het algebraïsch datatype dat door GHC gebruikt wordt:

```
type CoreProgram = [Bind Var]
data Bind b
  = NonRec b Expr
  | Rec [(b, Expr)]
data Expr b
  = Var Id
  | Lit Literal
  | App (Expr b) (Expr b)
  | Lam b (Expr b)
  | Let (Bind b) (Expr b)
  | Case (Expr b) b Type [Alt b]
  | Cast (Expr b) Coercion
  | Tick (Tickish Id) (Expr b)
  | Type Type
  | Coercion Coercion
type Alt b = (AltCon, [Id], Expr b)
```

Var stelt eenvoudigweg variabelen voor, en literals worden door *Lit* geconstrueerd. *App* en *Lam* zijn respectievelijk lambda-applicatie en lambda-abstractie, concepten waarmee we bekend zijn uit de lambda-calculus. *Let* stelt **let**-expressies voor, zowel recursief als niet-recursief. *Case* stelt **case**-expressies voor maar heeft meerdere parameters: een extra binder voor de expressie die onderzocht wordt door de **case**-expressie (ook de *scrutinee* genoemd), en het type van de resulterende alternatieven. *Cast*, *Tick*, *Type* en *Coercion* worden gebruikt voor expressies die niet relevant zijn voor het onderwerp van deze thesis. We gaan hier dus niet dieper op in.

Haskell-programma's worden door de compiler voorgesteld in een dergelijke abstracte syntaxboom, en plugins kunnen deze bomen manipuleren om de gewenste transformatie uit te voeren. De syntaxbomen voor de belangrijkste GHC Core-expressies worden getoond in Figuur 5.2.

x	*Var* `"x"`
2	*Lit* 2
$e_1\ e_2$	*App* $e_1\ e_2$
$\lambda x \to e$	*Lam x e*
let $x = e_1$ **in** e_2	*Let* (*NonRec x* e_1) e_2
case e_1 **of** $C\ x_1\ x_2 \to e_2$	*Case* e_1 _ _ $[(DataCon\ C, [x_1, x_2], e_2)]$

Figuur 5.2: Een overzicht van hoe GHC-Core expressies worden voorgesteld in de abstracte syntaxboom.

Ter illustratie beschouwen we een plugin pass die zorgt voor het inlinen van niet-recursieve binds. Een dergelijke pass zorgt dus voor een transformatie van **let** $x = e_1$ **in** e_2 naar $[x \mapsto e_1]e_2$.

Let op: om deze code eenvoudig te houden gaan we ervan uit dat alle variabelen uniek zijn over het gehelen programma, m.a.w. er kan geen shadowing optreden. In GHC is dit echter **niet het geval**, en in de praktijk moeten we dus voorzichtiger zijn als we een dergelijke plugin implementeren.

```
simpleBetaReduction :: CoreProgram → CoreM CoreProgram
simpleBetaReduction = return ∘ map (goBind [])
  where
    goBind :: [(Var, Expr Var)] → Bind Var → Bind Var
    goBind env (NonRec x e) = NonRec x (go ((x, e) : env) e)
    goBind env (Rec bs)     = Rec [(x, go env e) | (x, e) ← bs]

    go :: [(Var, Expr Var)] → Expr Var → Expr Var
    go env (Var x)                =
      case lookup x env of Nothing → Var x; Just e → (go env e)
    go env (Lit x)                = Lit x
    go env (App e1 e2)            = App (go env e1) (go env e2)
    go env (Lam x e)              = Lam x (go env e)
    go env (Let (NonRec x e1) e2) = go ((x, e1) : env) e2
    go env (Let (Rec bs) e2)      =
      Let (Rec [(x, go env e1) | (x, e1) ← bs]) (go env e2)
    go env (Case e1 x1 ty alts)   =
      Case (go env e1) x1 ty
        [(ac, bnds, go env e2) | (ac, bnds, e2) ← alts]
    go env (Cast e c)             = Cast (go env e) c
    go env (Tick t e)             = Tick t (go env e)
    go env (Type t)               = Type t
    go env (Coercion c)           = Coercion c
```

Eenmaal een dergelijke plugin geschreven is, kan ze eenvoudig gebruikt worden. Hiervoor gaan we als volgt te werk. Eerst *packagen* we de plugin met *cabal* [18] en installeren we ze:

```
cabal install my-plugin
```

Vervolgens kan men door slechts enkele commandolijn-argumenten mee te geven GHC opdragen dat deze plugin geladen en uitgevoerd moet worden tijdens de compilatie:

```
ghc --make -package my-plugin -fplugin MyPlugin test.hs
```

Waarbij *MyPlugin* de module is die *plugin* :: *Plugin* bevat. `my-plugin` is de naam van het geïnstalleerde cabal-package. Dit toont aan dat het relatief

eenvoudig is om GHC uit te breiden of aan te passen met behulp van het plugin framework. Bovendien hoeven we geen GHC code aan te passen, zolang de vereiste transformaties op de abstracte syntaxbomen kunnen uitgevoerd worden.

5.3 De what-morphism plugin

Voor deze thesis ontwikkelden we een GHC plugin genaamd *what-morphism* [7]. Er zijn vier passes geïmplementeerd in deze plugin, alhoewel ze niet alle vier gebruikt worden.

- *WhatMorphism.Fold*: conversie van expliciete recursie naar een functie in termen van een fold.
- *WhatMorphism.Build*: herschrijven van functies die gebruik maken van expliciete constructoren, naar functies die een build te gebruiken.
- *WhatMorphism.Inliner*: een extra inliner die beter aanstuurbaar is in vergelijking met de GHC inliner.
- *WhatMorphism.Fusion*: een implementatie van de foldr/build-fusion die werkt voor alle datatypes zonder dat er extra `{-# RULES #-}` pragma's nodig zijn.

De passes werken op basis van een best-effort en kunnen dus falen voor bepaalde expressies. Dit betekent niet dat de compilatie wordt afgebroken, wel dat we de transformatie niet kunnen maken en dus de originele expressie behouden.

5.3.1 WhatMorphism.Fold

De *WhatMorphism.Fold* pass is een meer deterministische implementatie van de regels in sectie 3.2.

We gebruiken de volgende functie ter illustratie (hier voorgesteld als Core-expressie):

$$
\begin{aligned}
&\mathit{foldlTree} :: (a \to b \to a) \to a \to \mathit{Tree}\ b \to a \\
&\mathit{foldlTree} = \lambda f\ z0\ \mathit{tree} \to \\
&\quad \textbf{let}\ \mathit{go} = \lambda z \to \lambda t \to \textbf{case}\ t\ \textbf{of} \\
&\qquad \mathit{Leaf}\ x \qquad \to f\ z\ x
\end{aligned}
$$

```
    Branch l r → let z' = go z l in go z' r
in go z0 tree
```

Dit is een *left fold* over een *Tree* (gedefinieerd in sectie 2.3.

Net zoals een left fold over een lijst, *foldl* kan uitgedrukt worden in functie van *foldr*, kan dit ook voor andere algebraísch datatypes, zoals *Tree*.

Om folds te detecteren, is het nuttig om elke *Bind* in het programma te bestuderen. Dit laat ons toe om mogelijke folds te vinden zowel in *top-level* binds alsook in lokale **let**- of **where**-binds. In ons voorbeeld kunnen we de *go* uit de **let**-bind omvormen tot een fold.

We volgen de volgende stappen om de recursieve *go* om te vormen tot een expressie die gebruikt maakt van een fold:

1. We beginnen met alle argumenten van de bind te verzamelen in een lijst en we kijken of er dan een *Case* volgt in de syntaxboom. We kunnen nu de argumenten rangschikken als volgt: het *scrutinee*-argument (het argument dat wordt afgebroken door de *Case*, type-argumenten, en bijkomende argumenten.

   De bijkomende argumenten partitioneren we in twee klasses: veranderlijke en statische argumenten. Een statisch argument is een argument dat hetzelfde is in elke oproep, zoals we eerder in sectie 3.2 bespraken. Type-argumenten dienen we ook op een andere manier te behandelen, maar hier gaan we niet dieper op in.

   In ons voorbeeld vinden we dat de boom t het scrutinee-argument is, en z een veranderlijk bijkomend argument.

2. In de fold zal het niet meer mogelijk zijn om rechtstreeks te verwijzen naar t. Daarom vervangen we in de rechterleden van de *Case*-alternatieven telkens t door het linkerlid van het alternatief. Voor *go* hebben we dus bijvoorbeeld voor het eerste alternatief $[t \mapsto Leaf\ x](f\ z\ x)$.

3. Vervolgens bestuderen we de expressies in de rechterleden van de verschillende *Case*-alternatieven. Het is de bedoeling deze alternatieven te herschrijven naar anonieme functies, zodat ze als argumenten voor de fold kunnen dienen.

   De argumenten voor deze anonieme functies zijn de binders van het alternatief gevolgd door de veranderlijke bijkomende argumenten. Zo krijgen we in ons voorbeeld $\lambda x\ z \to \ldots$ en $\lambda l\_rec\ r\_rec\ z \to \ldots$[2].

[2]Het *_rec*-suffix duidt hier op het feit dat dit niet de originele binders zijn, aangezien het type veranderde. Dit is een implementatie-detail, dat verder geen invloed heeft op de essentie van het algoritme.

We construeren dan verder deze anonieme functies door te vertrekken vanuit de rechterleden van de alternatieven en hierin expliciete recursie te elimineren. Wanneer we zo'n expliciete recursie vinden, kijken we welk argument er op de plaats van de scrutinee staat.

Als onze functie daadwerkelijk een fold is, zal dit altijd een recursieve subterm van het datatype zijn: een fold zal altijd de recursieve subtermen op een recursieve manier reduceren. Indien er een ander argument op de plaats van de scrutinee staat, kunnen we het algoritme stopzetten, omdat de functie geen fold is. Anders herschrijven we de recursieve oproep als de nieuwe binder voor de recursieve subterm toegepast op de veranderlijke bijkomende argumenten.

In ons voorbeeld krijgen we dus:

$$
\begin{array}{lcl}
\mathit{Leaf}\ x \quad \to f\ z\ x & & \lambda x\ z \quad \to f\ z\ x \\
\mathit{Branch}\ l\ r \to & \Leftrightarrow & \lambda \mathit{l\_rec}\ \mathit{r\_rec}\ z \to \\
\quad \textbf{let}\ z' = \mathit{go}\ z\ l\ \textbf{in}\ \mathit{go}\ z'\ r & & \quad \textbf{let}\ z' = \mathit{l\_rec}\ z\ \textbf{in}\ \mathit{r\_rec}\ z'
\end{array}
$$

We zien aldus hoe op deze manier veranderlijke bijkomende argumenten als z worden doorgegeven.

4. Tenslotte dienen we de anonieme functies aan de argumenten van de fold te koppelen: dat doen we in de implementatie simpelweg door de volgorde van de constructoren op te vragen en te herordenen naar de volgorde van de argumenten van de fold (*foldTree* in dit geval). We geven natuurlijk ook de scrutinee mee als laatste argument voor deze fold, gevolgd door de bijkomende argumenten, zodat deze kunnen worden doorgegeven aan verdere oproepen.

   We krijgen dus:

```
foldlTree' :: (a -> b -> a) -> a -> Tree b -> a
foldlTree' = λf z0 tree ->
  foldTree
    (λx z -> f z x)
    (λl_rec r_rec z -> let z' = l_rec z in r_rec z')
    tree
    z0
```

5.3.2 WhatMorphism.Build

WhatMorphism.Build is de pass die ervoor verantwoordelijk is om functies die waarden construeren met concrete constructoren, om te zetten naar

functies die gebruik maken van de build voor het corresponderende datatype.

We gebruiken ook hier ook meer deterministisch algoritme dan de voorgestelde niet-deterministische regels voorgesteld in hoofdstuk 4. Als voorbeeld gebruiken we de functie *infiniteTree*:

$$
\begin{aligned}
&\mathit{infiniteTree} :: \mathit{Tree\ Int} \\
&\mathit{infiniteTree} = \\
&\quad \textbf{let}\ \mathit{go} = \lambda n \to \mathit{Branch}\ (\mathit{Leaf}\ n)\ (\mathit{go}\ (n+1)) \\
&\quad \textbf{in}\ \mathit{go}\ 1
\end{aligned}
$$

We zoeken overal naar functies die we kunnen omzetten, dus zowel in top-level definities (*infiniteTree*) als lokale definities (*go*). In dit geval kan *go* geschreven worden in termen van *buildTree*. Het algoritme verloopt als volgt:

1. We kijken naar het type van de functie in kwestie en bepalen aan hand daarvan het return-type. In dit geval krijgen hebben we $\mathit{go} :: \mathit{Int} \to \mathit{Tree\ Int}$ en dus is *Tree Int* ons return-type.

   Let erop dat zoals we in sectie 2.1 opmerkten, Haskell type-signaturen op verschillende manieren kunnen gelezen worden. Bij een functie als $f :: \mathit{Int} \to \mathit{Int} \to \mathit{Tree\ Int}$ kunnen we zowel $\mathit{Int} \to \mathit{Tree\ Int}$, *Tree Int* (en in principe ook $\mathit{Int} \to \mathit{Int} \to \mathit{Tree\ Int}$) als type-signatuur beschouwen. Dit vormt echter geen probleem: aangezien er geen build bestaat voor functietypes (deze types hebben geen constructoren) dienen we altijd voor *Tree Int* te kiezen.

2. In subsectie 5.3.5 bespraken we de annotaties die GHC toelaat voor types. Hierdoor is het op dit moment mogelijk om, naast allerlei info over het datatype (welke constructoren zijn er?) ook de bijhorende build-functie op te vragen.

   We maken nu een nieuwe functie *g* aan die dezelfde argumenten neemt als de functie in kwestie maar een meer generiek return-type heeft: een vrije variabele *b*. In ons voorbeeld:

   $$
   \begin{aligned}
   &\mathit{go} :: \mathit{Int} \to \mathit{Tree\ Int} \\
   &g \ \ :: \mathit{Int} \to b
   \end{aligned}
   $$

   Uit de type-signatuur van de build kunnen we vervolgens de verschillende argumenten afleiden:

   $$
   \begin{aligned}
   &\mathit{leaf} \ \ :: a \to b \\
   &\mathit{branch} :: b \to b \to b
   \end{aligned}
   $$

Op dit moment hebben we dus genoeg informatie om een skelet-functie te construeren die er voor ons voorbeeld uitziet als:

$$
\begin{aligned}
&\mathit{infiniteTree} :: \mathit{Tree\ Int} \\
&\mathit{infiniteTree} = \\
&\quad \textbf{let}\ \mathit{go} = \lambda n \to \mathit{buildTree}\ \$\ \lambda \mathit{leaf}\ \mathit{branch} \to \\
&\qquad \textbf{let}\ g = \lambda n' \to \ldots \\
&\qquad \textbf{in}\ g\ n \\
&\quad \textbf{in}\ \mathit{go}\ 1
\end{aligned}
$$

3. We kunnen nu afdalen in de definitie van *go* en deze herschrijven naar de functie *g*. Om dit te doen bestuderen we de return-waarde van *go*. We onderscheiden drie gevallen:

   - Er is sprake van directe recursie naar *go*. Dit kunnen we toelaten, al moeten we dit natuurlijk herschrijven naar directe recursie naar *g* zodat de functie well-typed blijft.
   - De return-waarde is een oproep naar de build-functie. Dit is uiteraard toegelaten, aangezien deze return-waarde ook de abstracte versies van de constructoren zal gebruiken. We dienen er wel voor te zorgen dat dezelfde variabelen gebruikt worden, hiertoe geven we ze gewoon als argumenten aan de g' van de geneste build. M.a.w., we herschrijven bijvoorbeeld:

     $$\mathit{build}\ \$\ \lambda \mathit{leaf}\ \mathit{branch} \to \mathit{build}\ g'$$

     Naar:

     $$\mathit{build}\ \$\ \lambda \mathit{leaf}\ \mathit{branch} \to g'\ \mathit{leaf}\ \mathit{branch}$$

   - Tenslotte is het uiteraard ook toegelaten de return-waarde te construeren met de concrete constructoren (in ons voorbeeld *Leaf* en *Branch*). Deze vervangen we dan door de abstracte versies (in ons voorbeeld *leaf* en *branch*). We moeten echter wel opletten als een constructor direct recursieve subtermen bevat (bijvoorbeeld *Branch*): daarbij passen we dezelfde drie gevallen opnieuw toe, maar dan op de subterm.

   Na toepassing van deze regels krijgen we:

$$
\begin{aligned}
&\mathit{infiniteTree} :: \mathit{Tree\ Int} \\
&\mathit{infiniteTree} = \\
&\quad \textbf{let}\ \mathit{go} = \lambda n \to \mathit{buildTree}\ \$\ \lambda \mathit{leaf}\ \mathit{branch} \to \\
&\qquad \textbf{let}\ g = \lambda n' \to \mathit{branch}\ (\mathit{leaf}\ n')\ (g\ (n' + 1)) \\
&\qquad \textbf{in}\ g\ n \\
&\quad \textbf{in}\ \mathit{go}\ 1
\end{aligned}
$$

   Met *go* een functie die herschreven is zodanig dat deze eventueel later kan genieten van foldr/build-fusion.

5.3.3 WhatMorphism.Inliner

Zoals we later ook in subsectie 5.4.2 zullen zien, is het niet altijd eenvoudig om te beslissen of een functie al dan niet moet worden ge-inlined.

Daarom implementeerden we eerst een eigen inliner die alle functies die we reeds omgezet hebben altijd inlinet. Dit bleek echter niet altijd tot goede resultaten te leiden, zoals ook in subsectie 5.4.2 te lezen is. Uiteindelijk kozen we er dus voor om zo goed mogelijk te proberen samenwerken met de GHC inliner via de pragma's die beschikbaar zijn.

5.3.4 WhatMorphism.Fusion

Zoals we reeds in subsectie 2.4.2 zagen, bestaat foldr/build-fusion eruit door met de volgende regel het patroon in de linkerlid van de stelling te vervangen door het patroon in het rechterlid:

$$\mathit{foldr}\ \mathit{cons}\ \mathit{nil}\ (\mathit{build}\ g) \equiv g\ \mathit{cons}\ \mathit{nil}$$

We kunnen dit doen met behulp van een `{-# RULES #-}` pragma. Dit ziet er dan als volgt uit:

```
{-# RULES "foldr/build-fusion"
    forall c n (g :: forall b. (a -> b -> b) b -> -> b).
    foldHaskellList c n (buildHaskellList g) = g c n #-}
```

Een dergelijke regel moet worden toegevoegd voor *elk* datatype waarvoor we fusion willen:

```
{-# RULES "foldTree/buildTree-fusion"
    forall l n (g :: forall b. (a -> b) -> (b -> b -> b) -> b) .
    foldTree l n (buildTree g) = g l n #-}
```

Door het expliciet gekwantificeerde type van *g* zijn deze regels erg verboos. Om dit te verhinderen stellen we twee mogelijkheden voor:

- Een Template Haskell functie die het `{-# RULES #-}` pragma genereert;
- Een extra pass, *WhatMorphism.Fusion*, die het fusion-patroon op een generieke manier implementeerd.

We implementeerden beide oplossingen. De Template Haskell functie kan als volgt aangeroepen worden:

```
$(deriveFold ''Tree "foldTree" "buildTree")
```

Deze genereert dan de bovenstaande `"foldTree/buildTree-fusion"` regel.

De *WhatMorphism.Fusion* pass neemt een andere aanpak. Door gebruik te maken van de reeds aanwezige annotaties (zie subsectie 5.3.5), kunnen we voor elke variabele die gebruikt wordt eenvoudig nagaan of dit al dan niet een fold of een build is voor een bepaald algebraïsch datatype.

Het concrete algoritme gaat als volgt:

1. We doorzoeken alle expressies naar variabelen waarvan we weten dat ze een fold zijn voor een bepaald algebraïsch datatype.
2. Vervolgens kunnen we de nodige informatie ophalen over dit datatype. Zo dienen we te weten hoeveel constructor-argumenten de fold heeft. De constructor-argumenten worden gevolgd door de waarde waarover de fold loopt. Als we niet genoeg argumenten hebben, stoppen we op dit moment met het algoritme.
3. We kijken of het laatste argument van de fold (de waarde waarover de fold loopt) een build is voor hetzelfde datatype. Als dit het geval is, hoeven we nu slechts het *g*-argument van de build te nemen, en dit vervolgens toepassen op de constructor-argumenten van de fold.

Alhoewel beide aanpakken min of meer hetzelfde doen, kiezen we er voor om de tweede aanpak, een *WhatMorphism.Fusion* pass te gebruiken. Hierover hebben we immers iets meer controle, zo breidden we deze pass al uit zodanig dat er door **let**-bindings kan gekeken worden. Ook kunnen we op deze manier voor iets meer debug-output zorgen waardoor we eenvoudiger kunnen zien waarom de fusion wel of niet wordt toegepast.

5.3.5 Annotaties

Een andere belangrijke, nieuwe feature van GHC die we gebruiken zijn *annotaties* [30]. Deze laten toe om extra informatie toe te voegen aan functies, types en modules. Dit is een bekend concept en wordt ook gebruikt in andere programmeertalen zoals Java [24].

Deze annotaties kunnen op verschillende manieren gebruikt worden, bijvoorbeeld:

- Informatie doorgeven over functies aan plugins;
- Extra documentatie of commentaar specificeren op een manier zodanig dat deze later kan opgevraagd worden door een andere tool;
- Bepaalde functies aanduiden als test cases, zoals gebeurt in de Java-bibliotheek JUnit [15].

Standaard-annotaties in GHC horen bij top-level functies of variabelen en zien er als volgt uit:

```
{-# ANN f "A String annotation" #-}
{-# ANN g [("arity", 3)]        #-}
```

We kunnen dus een top-level functie of variabele annoteren met een expressie *e* van om het even welk type[3].

We kunnen ook modules of types annoteren door gebruik te maken van de volgende syntax:

```
{-# ANN type T e #-}
{-# ANN module e #-}
```

In ons geval willen we algebraïsche datatypes koppelen aan de corresponderende folds en builds. Daarom gebruiken we een type-annotatie van het type *RegisterFoldBuild*.

data *RegisterFoldBuild* = *RegisterFoldBuild*
{ *registerFold* :: *String*
, *registerBuild* :: *String*
} **deriving** (*Data*, *Show*, *Typeable*)

Dit datatype bevat simpelweg de namen van de corresponderende fold en build van een datatype en wordt op de volgende manier geassocieerd met het type:

```
{-# ANN type Tree (RegisterFoldBuild "foldTree" "buildTree") #-}
```

Eens deze annotaties aanwezig zijn in de source code, kunnen we ze op eenvoudige wijze ophalen in onze plugin wanneer we deze informatie nodig hebben.

[3]Dit type moet wel serialiseerbaar zijn. Hiertoe wordt de generische *Data.Data* klasse [21] gebruikt.

5.3.6 Detectie of transformatie?

Bij installatie is het mogelijk een aantal opties in te stellen. Hier is het mogelijk om in te stellen of we folds en builds enkel willen *detecteren* of ook effectief *transformeren*.

De detectiemode is zeer nuttig aangezien deze zonder meer op bestaande code kan uitgevoerd worden, zonder dat deze gewijzigd moet worden.

Voor de transformatiemode is dit wel nodig: dan moeten we namelijk (voorlopig manueel):

- Imports toevoegen zoals de module *WhatMorphism.HaskellList* (zodanig dat onze *foldr* en *build* functies voor lijsten in beschikbaar zijn), *WhatMorphism.TemplateHaskell* (zodat de Template Haskell *deriveFold* en *deriveBuild* beschikbaar zijn) en tenslotte importeren we ook de module *WhatMorphism.Annotations* (om annotaties te kunnen toevoegen, zie subsectie 5.3.5).
- Als we in een module builds en folds willen genereren, moeten we ook in de module-header de pragma's `{-# LANGUAGE Rank2Types #-}` en `{-# LANGUAGE TemplateHaskell #-}` toevoegen. Hiervan dient het eerste om het expliciet universeel gekwantificeerde type van build-functies toe te laten, en het tweede laat ons toe de *deriveFold* en *deriveBuild* functie op te roepen.
- Bij types waarvoor we een fold en build willen genereren plaatsen we vervolgens de *deriveFold*- en *deriveBuild*-functies, en ook een annotatie.

5.4 Aanpassen van de compilatie-passes

5.4.1 Volgorde van de passes

Zoals eerder besproken in sectie 5.2, kan onze plugin, op het moment dat deze geladen wordt, de passes die GHC zal uitvoeren wijzigen. We kunnen natuurlijk bijvoorbeeld naïef onze plugins als eerste runnen, maar om goede resultaten te boeken, blijkt het uitermate belangrijk de plugins optimaal te laten samenwerken met GHC.

Ten eerste willen we geen enkele GHC-phase verwijderen: anders beginnen we onmiddelijk met een nadeel tegenover de standaard lijst van passes. Om

maximaal resultaat te boeken, runnen we onze reeks plugins telkens tussen elke twee GHC passes, en wel in deze volgorde:

1. Eerst voeren we *WhatMorphism.Build* uit. Het is belangrijk dat we eerst functies naar builds omzetten en vervolgens pas naar folds. Beschouw het volgende voorbeeld om dit te illustreren:

   $$\begin{aligned}&\mathit{upper} :: \mathit{String} \to \mathit{String}\\&\mathit{upper}\ [\,] \quad\ = [\,]\\&\mathit{upper}\ (x : xs) = \mathit{toUpper}\ x : \mathit{upper}\ xs\end{aligned}$$

   We kunnen dit eerst naar een build omzetten met ons algoritme:

   $$\begin{aligned}&\mathit{upper} :: \mathit{String} \to \mathit{String}\\&\mathit{upper}\ \mathit{str} = \mathit{build}\ \$\ \lambda \mathit{cons}\ \mathit{nil} \to\\&\quad \textbf{let}\ g\ \mathit{str}' = \textbf{case}\ \mathit{str}'\ \textbf{of}\\&\qquad [\,] \quad\ \to \mathit{nil}\\&\qquad (x : xs) \to \mathit{cons}\ (\mathit{toUpper}\ x)\ (g\ xs)\\&\quad \textbf{in}\ g\ \mathit{str}\end{aligned}$$

   En vervolgens naar een fold:

   $$\begin{aligned}&\mathit{upper} :: \mathit{String} \to \mathit{String}\\&\mathit{upper}\ \mathit{str} = \mathit{build}\ \$\ \lambda \mathit{cons}\ \mathit{nil} \to\\&\quad \textbf{let}\ g\ \mathit{str}' = \mathit{foldr}\ (\lambda x\ xs \to \mathit{cons}\ (\mathit{toUpper}\ x)\ xs)\ \mathit{nil}\ \mathit{str}'\\&\quad \textbf{in}\ g\ \mathit{str}\end{aligned}$$

   Beide omzettingen zijn succesvol en nu kan de functie *upper* zowel als producent als consument van een lijst van foldr/build-fusion genieten.

   Stel dat we echter eerst *WhatMorphism.Fold* zouden uitvoeren:

   $$\begin{aligned}&\mathit{upper} :: \mathit{String} \to \mathit{String}\\&\mathit{upper} = \mathit{foldr}\ (\lambda x\ xs \to \mathit{toUpper}\ x : xs)\ [\,]\end{aligned}$$

   Dit werkt, maar vervolgens zal *WhatMorphism.Build* dit niet meer kunnen omzetten naar een build: ons algoritme is niet in staat om te zien dat *foldr* enkel de constructoren $(:)$ en $[\,]$ zal gebruiken. In dit geval kan *upper* dus enkel als consument van een lijst van foldr/build-fusion genieten.

   We kunnen concluderen dat het voordelig *WhatMorphism.Build* uit te voeren voor *WhatMorphism.Fold* en niet omgekeerd.

   Een alternatief zou zijn om een extra B-FOLD regel toe te voegen aan de regels in hoofdstuk 4. Deze zou dan ook bepaalde soorten folds (waar de resultaatwaarde wordt opgebouwd met behulp van bepaalde

constructoren) kunnen omzetten naar builds. Bij ons voorbeeld *upper* zouden we dan het volgende resultaat krijgen na omzetting naar een functie die gebruik maakt van *build*:

$$upper :: String \to String$$
$$upper = build\ \$\ \lambda cons\ nil \to foldr\ (\lambda x\ xs \to cons\ (toUpper\ x)\ xs)\ nil$$

2. Vervolgens voeren we *WhatMorphism.Fold* uit, vanwege de redenen die we hierboven uitlegden.

3. Voor functies die we succesvol omzetten naar folds en builds, passen we de *inliner-info* aan. Dit zorgt ervoor dat GHC deze agressief zal proberen inlinen. Dit is nodig om aan foldr/build-fusion te doen, aangezien deze pass enkel de fold- en build-functies herkent, en niet bijvoorbeeld functies geschreven in termen van fold en build.

   Het is dus geen goed idee om nu al *WhatMorphism.Fusion* uit te voeren, aangezien het zeer onwaarschijnlijk is dat deze iets zal kunnen herkennen.

   In plaats daarvan voeren we nu dus een bijkomende *Simplifier* pass uit met $sm\_inline = True$: dit geeft een goede kans om nodige functies te inlinen.

4. Tenslotte kunnen we de *WhatMorphism.Fusion* pass uitvoeren.

5.4.2 Inlinen of niet inlinen?

Een andere belangrijke vraag is of we de functies *foldr* en *build* (en natuurlijk de andere folds en builds die we genereren voor bijkomende algebraïsche datatypes) willen inlinen. Het antwoord is geen eenvoudige ja of nee, aangezien beide keuzes voordelen en nadelen hebben.

Indien we de functies niet zouden inlinen, met een `{-# NOINLINE #-}` pragma, kunnen we meer foldr/build-fusion instanties detecteren: zodra één van deze functies ge-inlined wordt, voldoen ze immers niet meer aan het patroon dat we herkennen.

Inlinen met een `{-# INLINE #-}` pragma zorgt echter wel voor snellere code omdat de functiecall-overhead vermeden wordt.

Het is dus best om te zoeken naar een middenweg hier. Gelukkig laat GHC voor `{-# INLINE #-}` pragma's *phase control* toe, wat wil zeggen dat we meer specifiek kunnen opgeven wanneer GHC moet (of mag) proberen een functie te inlinen. Om dit te doen, gebruikt GHC een nummering van phases: deze

loopt af naar 0 (de laatste phase). In een `{-# INLINE #-}` pragma kan men vervolgens dergelijke phases specificeren: Tabel 5.1 geeft een overzicht van de verschillende mogelijkheden.

	Voor phase n	**Phase** n **en later**
Geen pragma	?	?
`{-# INLINE f #-}`	✓	✓
`{-# NOINLINE f #-}`	×	×
`{-# INLINE [n] f #-}`	×	✓
`{-# INLINE [~n] f #-}`	✓	×
`{-# NOINLINE [n] f #-}`	×	?
`{-# NOINLINE [~n] f #-}`	?	×

Tabel 5.1: Een overzicht van de verschillende `{-# INLINE #-}` *pragma's en of ze de functie f al dan niet inlinen. Bij een ? beslist GHC zelf op basis van een groot aantal heuristieken.*

We kiezen dus voor de volgende pragma's voor elke fold en build, zowel voor lijsten als andere algebraïsche datatypes:

```
{-# INLINE [0] fold  #-}
{-# INLINE [0] build #-}
```

Aangezien phase 0 de laatste phase is, krijgen we zo beide voordelen: voor de laatste phase worden deze functies nooit ge-inlined, wat foldr/build-fusion haalbaarder maakt. In de laatste phase worden ze wel ge-inlined, en dus wordt ook de functiecall-overhead vermeden als er geen mogelijkheid werd gevonden tot foldr/build-fusion.

Deze pragma's wordt ook automatisch gegenereerd door onze Template Haskell code. De programmeur hoeft hier dus niet over na te denken.

6 Evaluatie

6.1 Detectie van folds

Een eerste aspect dat we kunnen bekijken is hoe goed de detectie van folds (zie hoofdstuk 3) werkt. We bespraken reeds dat onze tool niet alle mogelijk folds kan detecteren. Helaas is het ook zeer intensief werk om een exacte telling te doen van het aantal folds in een codebase, aangezien dit manueel moet gebeuren. Het lijkt dus bijzonder lastig om valse negatieven te vinden.

We kunnen wel vergelijken met andere tools. Hiervan is *HLint* [25] een voorbeeld. HLint is een tool dat Haskell-packages leest en suggesties geeft over de gebruikte code-stijl. Het focust dus op refactoring in plaats van optimalisaties en werkt rechtstreeks op de Haskell-code, waar wij kozen met de GHC Core te werken (zie sectie 5.1). Eén van de suggesties die HLint kan geven is het gebruik van *map*, *foldl* of *foldr* in plaats van expliciete recursie.

We toonden eerder al aan dat zowel *map* en *foldl* in termen van *foldr* uitgedrukt kunnen. Als we dus de som nemen van het aantal functies die herschreven kunnen worden als *map*, *foldl* of *foldr* volgens HLint, krijgen we dus het aantal folds over lijsten gedetecteerd door HLint.

We kunnen dit natuurlijk niet rechtstreeks vergelijken met het aantal folds dat wij detecteren in een Haskell-package: wij detecteren immers folds over alle algebraïsche datatypes. We maken dus een onderscheid tussen folds over lijsten en folds over andere algebraïsche datatypes.

Een overzicht van de resultaten is te zien in Tabel 6.1. We zien duidelijk dat we meer folds vinden dan HLint. Bovendien probeerden we onze tool ook uit op de testcases die meegeleverd worden – en deze worden allemaal herkent als folds. Dit duidt aan dat wij een strikte subset van mogelijk folds detecteren.

Het feit dat HLint geen enkele mogelijke fold kan vinden in sommige pack-

Package	**Totaal**	Lijst	ADT	V. arg.	N. rec.	**HLint**
Cabal-1.16.0.3	20	11	9	6	0	9
containers-0.5.2.1	100	11	89	41	11	1
cpphs-1.16	5	2	3	3	0	1
darcs-2.8.4	66	65	8	1	0	6
ghc-7.6.3	327	216	111	127	9	26
hakyll-4.2.2.0	5	1	4	3	0	0
haskell-src-exts-1.13.5	37	11	26	15	0	2
hlint-1.8.44	6	3	3	1	0	0
hscolour-1.20.3	4	4	0	0	0	2
HTTP-4000.2.8	6	6	0	2	0	3
pandoc-1.11.1	15	15	0	1	0	2
parsec-3.1.3	3	3	0	1	0	0
snap-core-0.9.3.1	4	3	1	1	0	0

Tabel 6.1: Een overzicht van het aantal gevonden folds in een aantal bekende packages.

ages suggereert ook dat de auteurs van deze packages misschien HLint gebruiken.

6.2 Detectie van builds

Naast de detectie van folds kunnen we ook de detectie van builds evalueren. Hier bestaat er naar ons weten echter geen vergelijkbare tool die dit soort detectie uitvoert. Dit betekent dat we niets hebben om onze resultaten mee te vergelijken.

De resultaten kunnen worden teruggevonden in Tabel 6.2. We kunnen concluderen dat de aantallen liggen in dezelfde grote-orde liggen als deze voor folds (zie sectie 6.1).

Merk op dat we hier de regel B-Build (zie hoofdstuk 4) niet gebruiken. Dit komt omdat we hier de optie gebruiken om enkel builds te detecteren en niet te transformeren (zie subsectie 5.3.6).

Package	**Totaal**	Lijst	ADT	Rec.
Cabal-1.16.0.3	101	81	20	5
containers-0.5.2.1	25	2	23	12
cpphs-1.16	6	5	1	3
darcs-2.8.4	354	354	0	26
ghc-7.6.3	480	178	302	53
hakyll-4.2.2.0	22	18	4	2
haskell-src-exts-1.13.5	140	74	66	16
hlint-1.8.44	69	62	7	1
hscolour-1.20.3	33	33	0	2
HTTP-4000.2.8	11	11	0	5
pandoc-1.11.1	97	97	0	16
parsec-3.1.3	10	10	0	0
snap-core-0.9.3.1	4	4	0	0

Tabel 6.2: Een overzicht van het aantal gevonden builds in een aantal bekende packages.

6.3 Foldr/build-fusion

Een telling maken van het aantal keer dat foldr/build-fusion kan toegepast worden in een Haskell package blijft echter een moeilijk probleem. We zitten met de volgende concrete moeilijkheden:

- Om foldr/build fusion te detecteren, dienen we eerst de transformaties naar folds en builds uit te voeren. Hiervoor is helaas nog manuele interventie nodig (zie subsectie 5.3.6).
- Veel code die cruciaal is voor de efficiëntie van een programma wordt momenteel manueel geoptimaliseerd en zal dus geen tijdelijke lijsten of andere algebraïsche datatypes aanmaken. Hier zijn dus geen mogelijkheden tot automatische optimalisatie.
- Een groot deel van de code wordt wel al geschreven in een stijl die gebruik maakt van hogere-orde functies. Omdat wij deze functies niet kunnen omzetten naar foldr/build-fusion[1], komen ze niet in aanmerking als kandidaten voor onze foldr/build-fusion. Code geschreven in

[1]Om meer precies te zijn, onze plugin kan wel herkennen dat een functie uit een externe library zoals bijvoorbeeld *map* een fold en een build is, en deze ook herschrijven, op voorwaarde dat deze functie eerst ge-inlined is. Maar dit gebeurt pas in een laat in de volgorde van de GHC-phases en op dat moment willen wij onze folds en builds ook inlinen (zie subsectie 5.4.2) – bijgevolg is er zijn er geen kansen om hierop nog fusie toe te passen.

hogere-orde stijl kan momenteel helaas dus niet samen gefused worden met code die wij vertaalden naar folds of builds.

In sectie 8.2 beschouwen we een aantal oplossingen voor deze problemen.

6.4 Tijdsmetingen

We onderzoeken nu de tijdswinsten die we kunnen behalen door foldr/build-fusion uit te voeren. Hiertoe maken we een lijst kleine programma's die fusable pijnlijnen van verschillende lengtes bevatten.

We beginnen met een aantal hulpfuncties voor lijsten te definiëren op expliciet recursieve wijze:

```
suml :: [Int] → Int
suml []       = 0
suml (x : xs) = x + suml xs

mapl :: (a → b) → [a] → [b]
mapl f = go
  where
    go []       = []
    go (x : xs) = f x : go xs

uptol :: Int → Int → [Int]
uptol lo up = go lo
  where
    go i
      | i > up       = []
      | otherwise = i : uptol (i + 1) up
```

We kunnen deze hulpfuncties ook definiëren voor ons *Tree*-type, opnieuw op expliciet recursieve wijze:

```
sumt :: Tree Int → Int
sumt (Leaf x)       = x
sumt (Branch l r) = sumt l + sumt r

mapt :: (a → b) → Tree a → Tree b
mapt f = go
  where
    go (Leaf x)       = Leaf (f x)
    go (Branch l r) = Branch (go l) (go r)

uptot :: Int → Int → Tree Int
```

```
uptot lo hi
    | lo >= hi      = Leaf lo
    | otherwise =
      let mid = (lo + hi) `div` 2
      in Branch (uptot lo mid) (uptot (mid + 1) hi)
```

Met deze hulpfuncties ter beschikking kunnen we een aantal pijplijnfuncties maken voor zowel lijsten als bomen, van variërende lengte:

```
l1, l2, l3, l4, l5 :: Int -> Int
l1 n = suml (1 `uptol` n)
l2 n = suml (mapl (+1) (1 `uptol` n))
l3 n = suml (mapl (+1) (mapl (+1) (1 `uptol` n)))
l4 n = ...
l5 n = ...


t1, t2, t3, t4, t5 :: Int -> Int
t1 n = sumt (1 `uptot` n)
t2 n = sumt (mapt (+1) (1 `uptot` n))
t3 n = sumt (mapt (+1) (mapt (+1) (1 `uptot` n)))
t4 n = ...
t5 n = ...
```

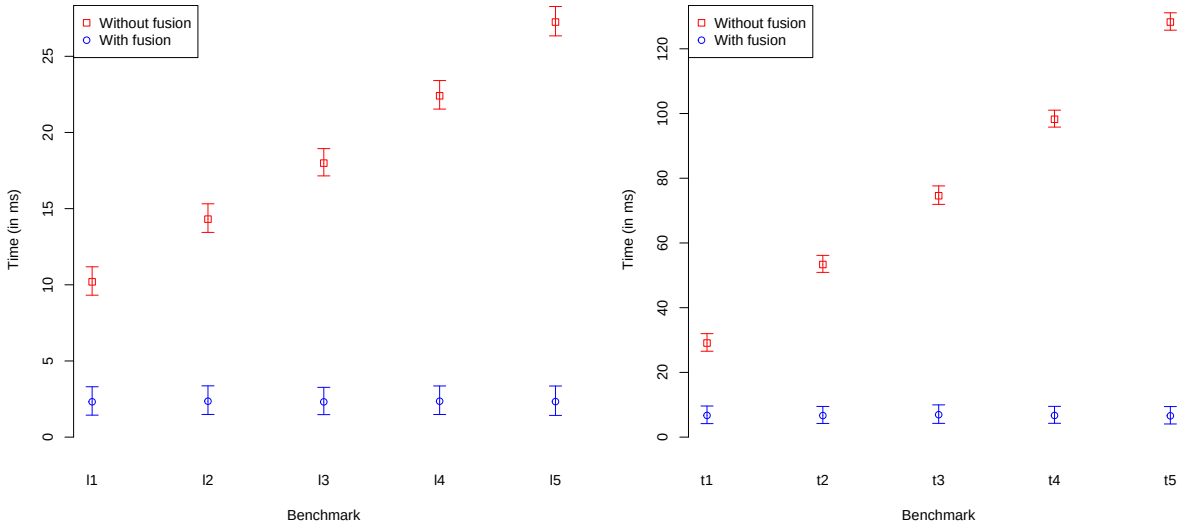
Deze functies zijn eenvoudig te benchmarken met behulp van de Criterion bibliotheek [26]. We gebruiken inputgrootte $n = 100000$ en voeren de benchmarks tweemaal uit: enerzijds met enkel de `-O2` compilatievlag, en anderzijds met de compilatievlaggen `-O2 -package what-morphism -fplugin WhatMorphism`.

De resultaten zijn te zien in Figuur 6.1 en Figuur 6.2. We zijn telkens geïnteresseerd in de versnelling, die we kunnen berekenen als:

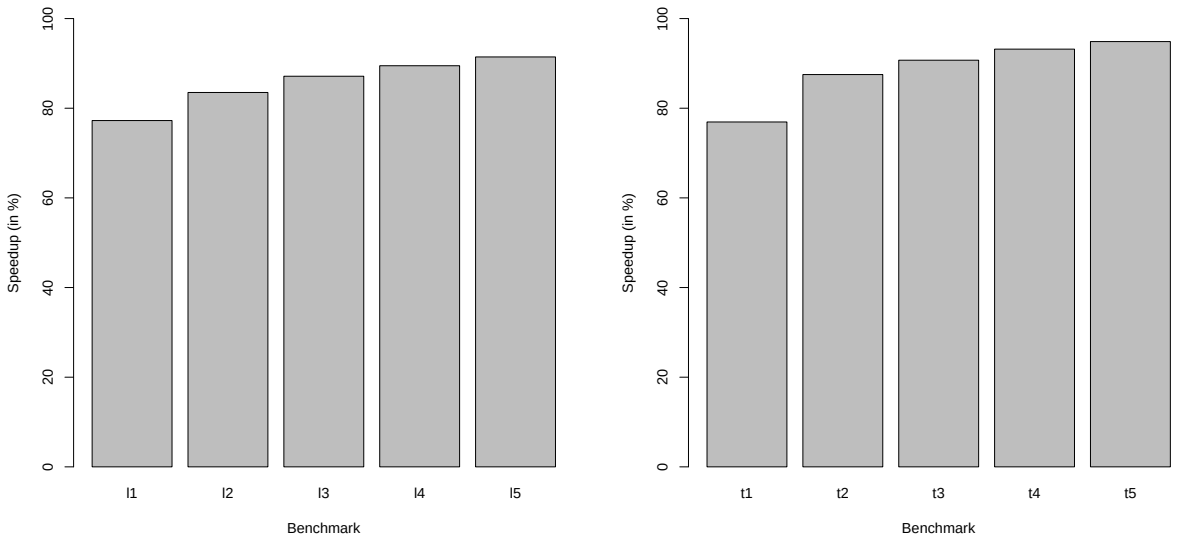
$$versnelling = \frac{t_2 - t_1}{t_2}$$

Met t_2 de tijdsmeting als we compileerden met *what-morphism* en t_1 de tijdsmeting met enkel de `-O2` vlag.

We zien dat we direct grote speedups krijgen bij *l0* en *t0*. Dit toont aan dat foldr/build-fusion ook voor heel kleine pipelines de moeite loont. Eveneens kunnen we uit de grafiek me relatieve resultaten afleiden dat de versnelling steeds dichter bij 100% zal komen naarmate de pijplijn langer wordt.



Figuur 6.1: De absolute resultaten van de tijdsmetingen voor lijsten (links) en bomen (rechts).



Figuur 6.2: De relatieve resultaten van de tijdsmetingen voor lijsten (links) en bomen (rechts).

7 Gerelateerd onderzoek

In het paper “A short cut to deforestation” [13] werd de foldr/build-fusion regel reeds besproken. De auteurs leggen de voordelen van deze aanpak uit, ondersteund door vele benchmarks. Ze vermelden echter ook de problemen rond het feit dat om van deze optimalisaties te kunnen genieten, alle programmeurs hun code in een specifieke stijl moeten schrijven. In dit concrete geval moet men gebruik te maken van *foldr* en *build* – anders is de optimalisatie niet van toepassing. Dit is natuurlijk precies het probleem dat we willen oplossen in deze thesis.

Stream fusion [5] is een geavanceerd alternatief op foldr/build-fusion. Een groot voordeel hiervan is dat het makkelijker fusion kan toepassen op functies als zoals *zip*.

Stream fusion werkt door lijsten voor te stellen als een tijdelijk type *Stream*:

```
data Stream a = ∀s.Stream (s → Step a s) s
data Step a s  = Done
               | Yield a s
               | Skip s
```

Lijsten kunnen worden omgezet van en naar een dergelijk *Stream* type:

```
stream :: [a] → Stream a
stream ls = Stream next ls
  where
    next []       = Done
    next (x : xs) = Yield x xs

unstream :: Stream a → [a]
unstream (Stream next s0) = unfold s0
  where
    unfold s = case next s of
      Done      → []
      Yield x s' → x : unfold s'
      Skip s'    → unfold s'
```

Nu dienen we functies als *map* te definiëren met behulp van het *Stream* type:

```
map :: (a → b) → [a] → [b]
map f = unstream ∘ map_s ∘ stream
  where
    map_s (Stream next s0) = Stream next' s0
      where
        next' s = case next s of
          Done      → Done
          Skip s'   → Skip s'
          Yield x s' → Yield (f x) s'
```

De hogere-orde functies zijn dus van de vorm $unstream \circ f_s \circ stream$. Als we hiervan een pijplijn maken krijgen we iets als bijvoorbeeld:

$$unstream \circ filter_s \circ stream \circ unstream \circ map_s \circ stream$$

Deze pijplijn kan geoptimaliseerd worden door stream fusion (het bewijs hiervan laten we achterwege):

Stelling 7.0.1.

$$stream \circ unstream \equiv id$$

Het nadeel van deze meer geavanceerde vorm van fusion is dus ook dat de programmeurs alle code in een specifieke stijl moeten schrijven (ditmaal in termen van *Stream*) om te kunnen genieten van deze optimalisatie. Hier is het dus ook interessant om te kijken of deze transformatie niet automatisch kan gebeuren, en onze thesis geeft hiervoor een basis.

Op een soortgelijke manier stelt Gibbons [12] voor om te programmeren in termen van folds en unfolds – een specifieke codestijl die hij *origami*-programmeren noemt. Unfolds zijn de tegenhanger van folds en kunnen gezien worden als een specifieke, gespecialiseerde versie van builds.

HLint [25] is een tool dat verschillende code-patronen kan herkennen en vervolgens suggesties geeft om deze code te verbeteren. Onder andere kan HLint ook bepaalde gevallen van expliciete recursie ontdekken en suggereren om deze code te herschrijven in termen van een hogere-orde functie zoals *map*, *foldr* of *foldl*. Zoals we al vermelden sectie 6.1, slaagt onze tool er in om meer van deze gevallen te ontdekken. Bovendien beperkt HLint zich tot lijsten en zoekt het niet naar recursiepatronen voor andere algebraïsche datatypes.

Sittampalam en de Moor ontwikkelden het MAG-framework [28], een semi-automatische aanpak om foldr-fusion uit te voeren. In deze aanpak moet de programmeur het zowel initiële programma specificeren, als het gewenste uiteindelijke programma (het *doelprogramma*) en een verzameling van herschrijfregels. Zo is er onder meer een herschrijfregel voor foldr-fusion:

$$
\begin{aligned}
& f\ (\mathit{foldr}\ c\ n\ l) = \mathit{foldr}\ c'\ n'\ l \\
& \quad \textbf{if}\ f\ n = n' \\
& \qquad \forall x\ y . f\ (c\ x\ y) = c'\ x\ (f\ y)
\end{aligned}
$$

Het MAG-framework probeert dan het doelprogramma af te leiden van het initiële programma door gebruik te maken van de opgegeven herschrijfregels. Nadien moet de programmeur nog nagaan of foldr-fusion enkel is toegepast voor strikte functies f – dit is een voorwaarde die immers niet opgegeven kan worden in het MAG-framework.

8 Conclusie en mogelijke uitbreidingen

8.1 Conclusie

Programmeurs verkiezen vaak om programmeertalen met een hoog abstractieniveau te gebruiken omwille van elegante, goed onderhoudbare code te kunnen schrijven. Toch blijft de prestatie van de uitgevoerde code van groot belang, m.a.w., de code moet vertaald kunnen worden naar efficiënte machinecode. De conflicten tussen beide doelstellingen kunnen heden ten dage vrij goed worden opgelost door het gebruik van geavanceerde compilers en slimme optimalisaties.

In deze thesis hebben we een dergelijke slimme optimalisatie beschreven en geïmplementeerd. Ons werk laat de programmeur toe zijn code op te bouwen met kleine, duidelijk begrijpbare en makkelijk testbare functies. Meer complexe operaties worden dan op een natuurlijke manier uitgedrukt als een combinatie van deze bouwstenen. Conceptueel maken deze combinaties tijdelijke structuren die vervolgens afgebroken worden om het eindresultaat te berekenen. Alhoewel dit toelaat om te redeneren over de code en het dus eenvoudiger wordt het programma te begrijpen, zal een naïeve compiler uiteindelijk bijzonder inefficiënte code afleveren die de beschikbare machinebronnen niet goed benut en bijgevolg zeer slecht presteert. Door ons werk kan een subset van deze functies echter op een automatische manier gefuseerd worden naar efficiëntere varianten.

Enerzijds hebben we aangetoond dat veel functies momenteel niet in aanmerking komen voor de optimalisaties die we hebben voorgesteld (zie secties 6.1 en 6.2); anderzijds hebben we aangetoond dat eenmaal fusie mogelijk is, de implementatie of resulterende machinecode veel efficiënter zal zijn tijdens het uitvoeren (zie sectie 6.4). Om de eerste tekortkoming aan te pakken hebben we een techniek ontwikkeld die toelaat om functies automatisch te herschrijven zodat ze wel in aanmerking komen voor deze optimalisaties.

Ons werk vormt dus een volgende stap in de implementatie van efficiënte hoog-niveau programmeertalen. Een artikel waarin dit werk wordt voorgesteld in een meer beknopte vorm, zal worden opgestuurd naar de ACM SIGPLAN Haskell Symposium 2013 conferentie onder de naam “Bringing Functions into the Fold”.

Tot slot vermelden we in de volgende secties nog een aantal uitbreidingen die in later onderzoek kunnen worden uitgewerkt.

8.2 Mogelijke uitbreidingen

In deze sectie geven we een aantal ideeën en suggesties over hoe ons werk kan uitgebreid worden in de toekomst.

8.2.1 Betere integratie

Zoals we reeds in sectie 6.3 vermeldden, werken onze plugins niet optimaal samen met bijvoorbeeld de bestaande *Data.List* module. Dit zorgt er bijvoorbeeld voor dat we nooit fusion krijgen in een pijplijn als:

$$f \circ map\ g$$

met f een expliciet recursieve functie die wij omgezet hebben naar een fold. Deze fusie is theoretisch natuurlijk wel mogelijk aangezien *map* een producent van een lijst is, en f een consument.

Er zijn hiervoor verschillende oplossingen. Een eerste is om de *Data.List* module te herschrijven in termen van onze functies, maar dit is niet echt praktisch. Een betere oplossing zou zijn om een aantal `{-# RULES #-}` pragma’s toe te voegen, één voor elke functie uit *Data.List*, zodanig dat deze ook gefused kunnen worden met onze folds en builds.

8.2.2 GADTs

Beschouw het volgende eenvoudige expressie-type:

```
data Expr = Const Int
          | Add Expr Expr
          | Equal Expr Expr
```

Dit type laat toe om numerieke waarden bij elkaar op te tellen en deze vervolgens te vergelijken. Met dit datatype is het is echter ook mogelijk om expressies op te stellen als:

$$Add\ (Equal\ (Const\ 1)\ (Const\ 2))\ (Const\ 3)$$

Deze expressie telt 3 op bij het resultaat van $1 \equiv 2$. In ongetypeerde talen geeft dit een runtime-fout. De vraag is nu natuurlijk of we met het geavanceerde typesysteem van Haskell een betere oplossing hiervoor kunnen construeren. Dit blijkt mogelijk te zijn door middel van GADTs [4], die toelaten toe het return-type van een constructor expliciet vast te leggen.

$$\begin{aligned}&\textbf{data}\ Expr\ a\ \textbf{where}\\ &\quad Const :: Int \to Expr\ Int\\ &\quad Add\ \ :: Expr\ Int \to Expr\ Int \to Expr\ Int\\ &\quad Equal :: Expr\ Int \to Expr\ Int \to Expr\ Bool\end{aligned}$$

Er werd reeds aangetoond [17] dat voor dergelijke GADTs ook folds en builds kunnen opgesteld worden. Bijgevolg is het wenselijk voor ons werk om ook in staat te zijn deze folds en builds te kunnen genereren en herkennen. Hiervan hebben we echter nog geen implementatie.

8.2.3 Indirect recursieve datatypes

In sectie 5.1 hadden we het reeds over indirecte recursie. Daarbij gaven we het voorbeeld:

$$\begin{aligned}&jibble :: [Int] \to Int\\ &jibble\ [\,] \quad\ = 1\\ &jibble\ (x : xs) = wiggle\ x\ xs\\ &wiggle :: Int \to [Int] \to Int\\ &wiggle\ x\ xs = x * jibble\ xs + 1\end{aligned}$$

Deze functie kunnen we niet onmiddelijk herkennen: dit kan pas na het inlinen van *wiggle* – maar we hoeven hier geen extra werk voor te verrichten.

Indirect recursieve algebraïsche datatypes vormen echter een moeilijker probleem. Een bekend voorbeeld van een indirect recursief datatype is de *Rose Tree* [1]. In Haskell kan een dergelijke boom voorgesteld worden als:

$$\textbf{data}\ Rose\ a = Rose\ a\ [Rose\ a]$$

Ook hier is het mogelijk om via inlinen het probleem te reduceren. We kunnen namelijk de constructoren van de lijst inlinen in de definitie van *Rose* [35]. Op die manier krijgen we:

```
data Node
data List
data Rose tag a where
  Node :: a → Rose List a → Rose Node a
  Nil  :: Rose List a
  Cons :: Rose Node a → Rose List a → Rose List a
```

In sectie 8.2.2 bespraken we reeds dat we ook folds en builds kunnen schrijven voor GADTs. Deze omzetting vormt dus één mogelijke oplossing, die echter niet echt praktisch is. We bekijken nu twee betere oplossingen. Het verschil ligt erin hoe we met het geneste type (in dit geval de lijst) omgaan.

Via een andere fold De eerste oplossing bestaat eruit ons algoritme uit te breiden zodat dat de subterm $[Rose\ a]$ herkend wordt als een ander type waarvoor een fold bestaat – in dit geval *foldr*. We geven dan de parameters voor *foldr* ook mee als argumenten aan *foldRose*:

```
foldRose :: (a → b → c)   -- De rose-constructor
         → (c → b → b)    -- De (:)-constructor
         → b              -- De []-constructor
         → Rose a → c
foldRose rose cons nil = go
  where
    go (Rose x ls) = rose x (foldr (cons ∘ go) nil ls)
```

Eveneens is het mogelijk om op deze manier een build te specificeren:

```
buildRose :: (∀c b.(a → b → c)
                → (c → b → b)
                → b
                → c)
              → Rose a
buildRose g = g Rose (:) []
```

En hieruit volgt dan de foldr/build-fusion regel voor het type *Rose* die we zien in Stelling 8.2.1 (zonder bewijs).

Stelling 8.2.1.

$$foldRose\ rose\ cons\ nil\ (buildRose\ g) \quad \equiv \quad g\ rose\ cons\ nil$$

Via de functie *fmap* Een nadeel van de vorige aanpak is dat deze enkel bruikbaar is als de manier waarom we de lijst consumeren *ook* een fold is. Er is ook een mogelijkheid waarin we niet met deze belemmering zitten, namelijk door gebruik te maken van de *Functor*-klasse [36]. Dit laat ons toe de recursieve subterm binnen de lijst te reduceren zonder de structuur van de lijst te kennen.

Dit geeft ons een veel eenvoudigere definitie van *foldRose*:

```
foldRose' :: (a → [b] → b) → Rose a → b
foldRose' rose = go
  where
    go (Rose x ls) = rose x (fmap go ls)
```

Ook voor *buildRose* krijgen we een eenvoudigere definitie:

```
buildRose' :: (∀b.(a → [b] → b) → b) → Rose a
buildRose' g = g Rose
```

Als we de twee aanpakken vergelijken hebben beide voordelen en nadelen. Een voordeel van de aanpak via *fmap* is dat we de lijst kunnen consumeren op eender welke manier – we zijn niet beperkt tot een *foldr*. Dit is echter een mes dat aan twee kanten snijdt: langs de andere kant betekent dit dat de consummatie van de lijst ook niet kan genieten van foldr/build-fusion.

Ook is niet elke indirect recursieve subterm een *Functor*: beschouw bijvoorbeeld de types *Expr* en *Decl*:

```
type Var  = String
data Expr = Const Int
          | Add Expr Expr
          | Mul Expr Expr
          | Var Var
          | Let Decl Expr
data Decl = Bind Var Expr
          | Seq Decl Decl
```

Noch *Expr* noch *Decl* is een *Functor* – hier behoort de aanpak via *fmap* dus niet tot de mogelijkheden.

Bibliografie

[1] Charles Blundell, Yee Whye Teh, and Katherine A Heller. Bayesian rose trees. *arXiv preprint arXiv:1203.3468*, 2012.

[2] Niklas Broberg. haskell-src-exts on hackage, 2008. URL `http://hackage.haskell.org/package/haskell-src-exts`.

[3] Felice Cardone and J Roger Hindley. History of lambda-calculus and combinatory logic. *Handbook of the History of Logic*, 5, 2006.

[4] James Cheney and Ralf Hinze. First-class phantom types. 2003.

[5] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion. from lists to streams to nothing at all. In *ICFP'07*, 2007.

[6] André L. de M. Santos. Compilation by transformation in non-strict functional languages, 1995.

[7] Jasper Van der Jeugt. The what-morphism ghc plugin source code, 2013. URL `https://github.ugent.be/javdrjeu/what-morphism`.

[8] Edsger W. Dijkstra. Go to statement considered harmful. *Comm. ACM*, 11(3):147–148, 1968. Letter to the Editor.

[9] Gilles Dubochet. Computer Code as a Medium for Human Communication: Are Programming Languages Improving? In Chris Exton and Jim Buckley, editors, *Proceedings of the 21st Working Conference on the Psychology of Programmers Interest Group*, pages 174–187, Limerick, Ireland, 2009. University of Limerick. ISBN 978-1-905952-16-8. URL `http://www.csis.ul.ie/PPIG09/`.

[10] Robert Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, number 19 in Proceedings of Symposia in Applied Mathematics, pages 19–32. American Mathematical Society, 1967.

[11] John E. Gaffney. Estimating the number of faults in code. *Software Engineering, IEEE Transactions on*, SE-10(4):459–464, 1984.

[12] Jeremy Gibbons et al. Origami programming. *The Fun of Programming. Palgrave*, 2003.

[13] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.

[14] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. ISSN 00029947. doi: 10.2307/1995158. URL http://dx.doi.org/10.2307/1995158.

[15] Andrew Hunt and David Thomas. *Pragmatic Unit Testing in Java with JUnit*. The Pragmatic Programmers, 2003.

[16] Graham Hutton. A tutorial on the universality and expressiveness of fold. *J. Funct. Program.*, 9(4):355–372, July 1999. ISSN 0956-7968. doi: 10.1017/S0956796899003500. URL http://dx.doi.org/10.1017/S0956796899003500.

[17] Patricia Johann and Neil Ghani. Foundations for structured programming with gadts. In *ACM SIGPLAN Notices*, volume 43, pages 297–308. ACM, 2008.

[18] Isaac Jones and Duncan Coutts. The haskell cabal: Common architecture for building applications and libraries, 2003. URL http://www.haskell.org/cabal/.

[19] Simon Peyton Jones. Wearing the hair shirt: a retrospective on haskell (invited talk). In *In ACM SIGPLAN Conferenge on Principles of Programming Languages (POPL'03)*, page 2003, 2003.

[20] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell Workshop*, volume 1, pages 203–233, 2001.

[21] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003*, pages 26–37. ACM Press, 2003.

[22] Simon Marlow et al. Haskell 2010 language report. *URL: http://www.haskell.org/definition/haskell2010.pdf*, 2010.

[23] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. pages 124–144. Springer-Verlag, 1991.

[24] Sun Microsystems. Annotations, 2011. URL http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html.

[25] Neil Mitchell. *HLint Manual*, 2006. URL http://community.haskell.org/~ndm/darcs/hlint/hlint.htm.

[26] Bryan O'Sullivan. Criterion, a new benchmarking library for haskell, 2009. URL http://www.serpentine.com/blog/2009/09/29/criterion-a-new-benchmarking-library-for-haskell/.

[27] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002. ISSN 0362-1340. doi: 10.1145/636517.636528. URL http://doi.acm.org/10.1145/636517.636528.

[28] Ganesh Sittampalam. Mag: A tool for transforming haskell programs. 1998.

[29] The GHC Team. The glasgow haskell compiler, 1989. URL http://www.haskell.org/ghc/.

[30] The GHC Team. The ghc wiki: Annotations, 2011. URL http://hackage.haskell.org/trac/ghc/wiki/Annotations.

[31] The GHC Team. Extending and using ghc as a library: Compiler plugins, 2011. URL http://hackage.haskell.org/trac/ghc/wiki/NewPlugins.

[32] Andrew Tolmach and Tim Chevalier. An external representation for the ghc core language. 2009.

[33] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.

[34] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231 – 248, 1990. ISSN 0304-3975.

[35] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ACM Sigplan Notices*, volume 44, pages 233–244. ACM, 2009.

[36] Brent Yorgey. Typeclassopedia. *The Monad.Reader*, (13):17–68, March 2009.